UNIT-1

Reasons for Studying of Programming Languages

Increased capacity to express ideas:

- People can easily express their ideas clearly in any language only when they have clear understanding of the natural language.
- Similarly, if programmers want to simulate the features of languages in another language, they should have some ideas regarding the concepts in other languages as well.

Improved background for choosing appropriate languages

- Many programmers when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to the project.

- If these programmers were familiar with a wider range of languages, they would be better able to choose the language that includes the features that best address the characteristics of the problem at hand.

Increased ability to learn new languages

- In software development, continuous learning is essential.
- The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only two or more languages.
- 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 the significance of implementation

- An understanding of implementation issues leads to an understanding of why languages are designed the way they are.
- This knowledge in turn leads to the ability to use a language more intelligently, as it was designed to use.
- We can become better programmers by understanding the choices among programming language constructs and consequences of those choices.

Better use of languages that are already known

- 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

- There is a global view of computing that can justify the study of programming language concepts.
- For example, many people believe it would have been better if ALGOL 60 had

displaced Fortran in the early 1960s, because it was more elegant and had much better control statements than Fortran. 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.

- If those who choose languages were better informed, perhaps, better languages would eventually squeeze out poorer ones.

Different Programming Domains

- **Scientific applications**

  – Large number of floating point computations. The most common data structures are arrays and matrices; the most common control structures are counting loops and selections

  – The first language for scientific applications was Fortran, ALGOL 60 and most of its descedants

  – Examples of languages best suited: Mathematica and Maple

- **Business applications**

  – Business languages are characterized by facilities for producing reports, precise ways of describing and storing decimal numbers and character data, and ability to specify decimal arithmetic operations.

  – Use decimal numbers and characters

  – COBOL is the first successful high-level language for those applications**.**

- **Artificial intelligence**

  – Symbols rather than numbers are typically manipulated

  – Symbolic computation is more conveniently done with linked lists of data rather than arrays

  – This kind of programming sometimes requires more flexibility than other programming domains

  – First AI language was LISP and is still most widely used

  – Alternate languages to LISP are Prolog – Clocksin and Mellish

- **Systems programming**

  – The operating system and all of the programming support tools of a computer system are collectively known as systems software

  – Need for efficiency because of continuous use

  – Low-level features for interfaces to external devices

  – C is extensively used for systems programming. The UNIX OS is written almost

entirely in C

- **Web software**
  - Markup languages
    - Such as XHTML
  - Scripting languages
    - A list of commands is placed in a file or in XHTML document for execution
    - Perl, JavaScript, PHP
- **Special-purpose languages**
  - RPG – business reports
  - APT – programmable machine tools

– GPSS – simulation

Language Evaluation Criteria

The following factors influences Language evalution criteria

1) Readability 2) Simplicity 3) Orthogonality 4) Writ ability 3) Reliability 4) Cost and 5) Others

Readability

- One of the most important criteria for judging a programming language is the ease with which programs can be read and understood.
- Language constructs were designed more from the point of view of the computer than of computer users
- From 1970s, the S/W life cycle concept was developed; coding was relegated to a much smaller role, and maintenance was recognized as a major part of the cycle, particularly in terms of cost. Because ease of maintenance is determined in large part by readability of programs, readability became an important measure of the quality of programs

Overall simplicity

- Language with too many features is more difficult to learn
- Feature multiplicity is bad. For example: In Java, increment can be performed if four ways as:
- Count= count+1
- Count+=1
- Count++
- ++count

- Next problem is operator overloading, in which single operator symbol has more than one meaning

Orthogonality

- A relatively small set of primitive constructs that can be combined in a relatively small number of ways
- Consistent set of rules for combining constructs (simplicity)
    - Every possible combination is legal
- For example, pointers should be able to point to any type of variable or data structure
- Makes the language easy to learn and read
- Meaning is context independent
- VAX assembly language and Ada are good examples
- Lack of orthogonality leads to exceptions to rules
- C is littered with special cases
    - E.g. - **struct**s can be returned from functions but arrays cannot

Orthogonolity is closely related to simplicity. The more orthogonal the design of a language, the fewer exceptions the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand.

- Useful control statements
- Ability to define data types and structures
- Syntax considerations
    - Provision for descriptive identifiers
        - BASIC once allowed only identifiers to consist of one character with an optional digit
    - Meaningful reserved words
    - Meaning should flow from appearance

Writability

- Most readability factors also apply to writability
- Simplicity and orthogonality
- Control statements, data types and structures
- Support for abstraction
    - Data abstraction

        - Process abstraction
    - Expressivity
        - It is easy to express program ideas in the language
        - APL is a good example
        - In C, the notation C++ is more convenient and shorter than C = C + 1
        - The inclusion of for statement in Java makes writing counting loops easier than the use of while
- **Reliability**
  A program is said to be reliable if performs to its specifications under all conditions.
    - Type checking
        - Type checking is simply testing for type errors in a given program, either by the compiler or during the program execution
        - Because run time type checking is expensive, compile time type checking is more desirable
        - Famous failure of space shuttle experiment due to **int / float** mix-up in parameter passing
    - Exception handling
        - Ability to intercept run-time errors
    - Aliasing
        - Ability to use different names to reference the same memory
        - A dangerous feature
    - Readability and writability both influence reliability
- **Cost**
    - Training programmers to use language
    - Writing programs in a particular problem domain
    - Compiling programs
    - Executing programs
    - Language implementation system (free?)
    - Reliability
    - Maintaining programs
- Others: portability, generality, well-definedness

Influences on Language Design

Computer Architecture

– Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture

• Programming Methodologies

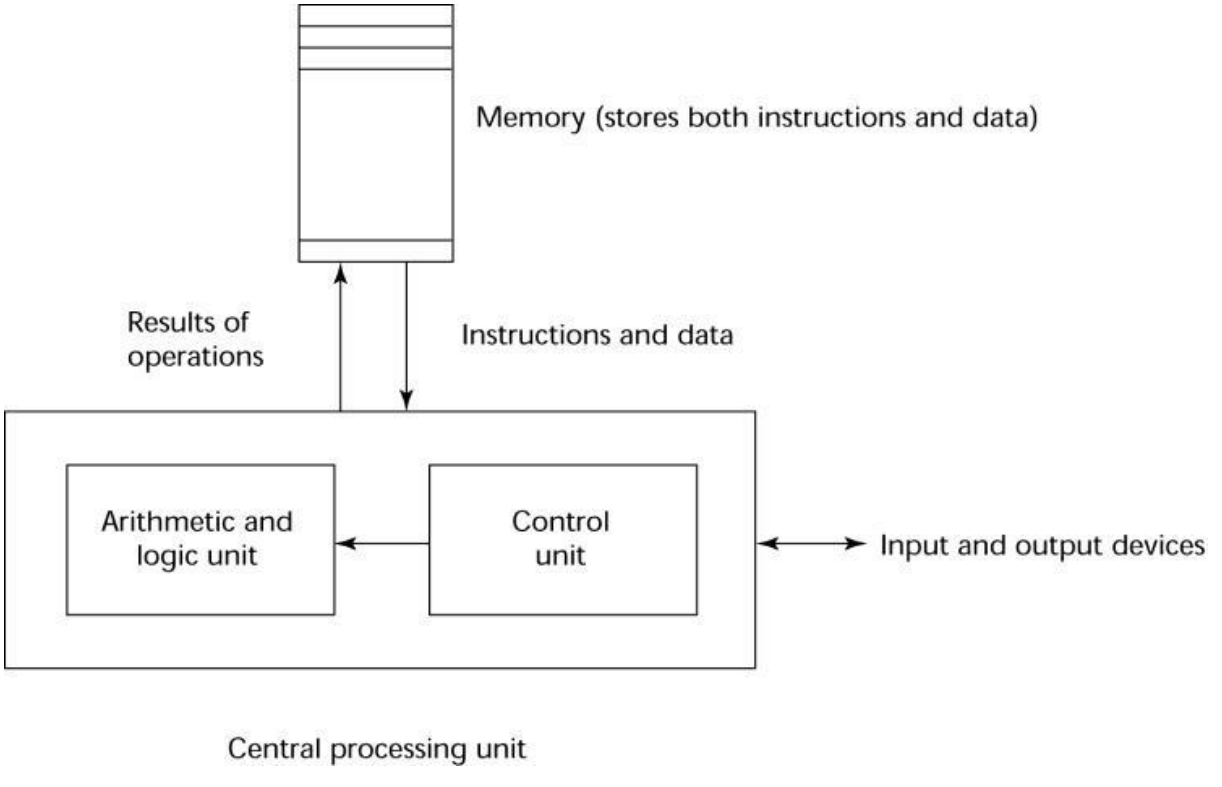
– New software development methodologies (e.g., object oriented software development) led to new programming paradigms and by extension, new programming languages.

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
- Variables model memory cells
- Assignment statements model piping

- Iteration is efficient

The von Neumann Architecture



Instruction Execution

Fetch-execute-cycle (on a von Neumann architecture)
initialize the program counter

repeat forever
fetch the instruction pointed by the counter
increment the counter **decode** the instruction **execute** the instruction end repeat

Programming Methodologies Influences

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures

  – structured programming
  – top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  – Data abstraction

- Middle 1980s: Object-oriented programming
  – Data abstraction + inheritance + polymorphism

Different types of programming languages and its design issues

Language Categories
• Imperative
  – Central features are variables, assignment statements, and iteration
  – Examples: C, Pascal

• Functional
  – Main means of making computations is by applying functions to given parameters
  – Examples: LISP, Scheme

• Logic (declarative)
  – Rule-based (rules are specified in no particular order)

  – Example: Prolog

• Object-oriented
  – Data abstraction, inheritance, late binding
  – Examples: Java, C++
• Markup
  – New; not a programming per se, but used to specify the layout of information in Web documents

– Examples: XHTML, XML

Language Design Trade-Offs

Reliability vs. cost of execution

– Conflicting criteria

– Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs

- **Readability vs. writability**
  – Another conflicting criteria
  – Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.
- **Writability (flexibility) vs. reliability**
  – Another conflicting criteria
  – Example: C++ pointers are powerful and very flexible but not reliably used

Different types Implementation Methods for programming languages.

We have three different types of implementation methods . They are

- Compilation
  – Programs are translated into machine language
- Pure Interpretation
  – Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  – A compromise between compilers and pure interpreters

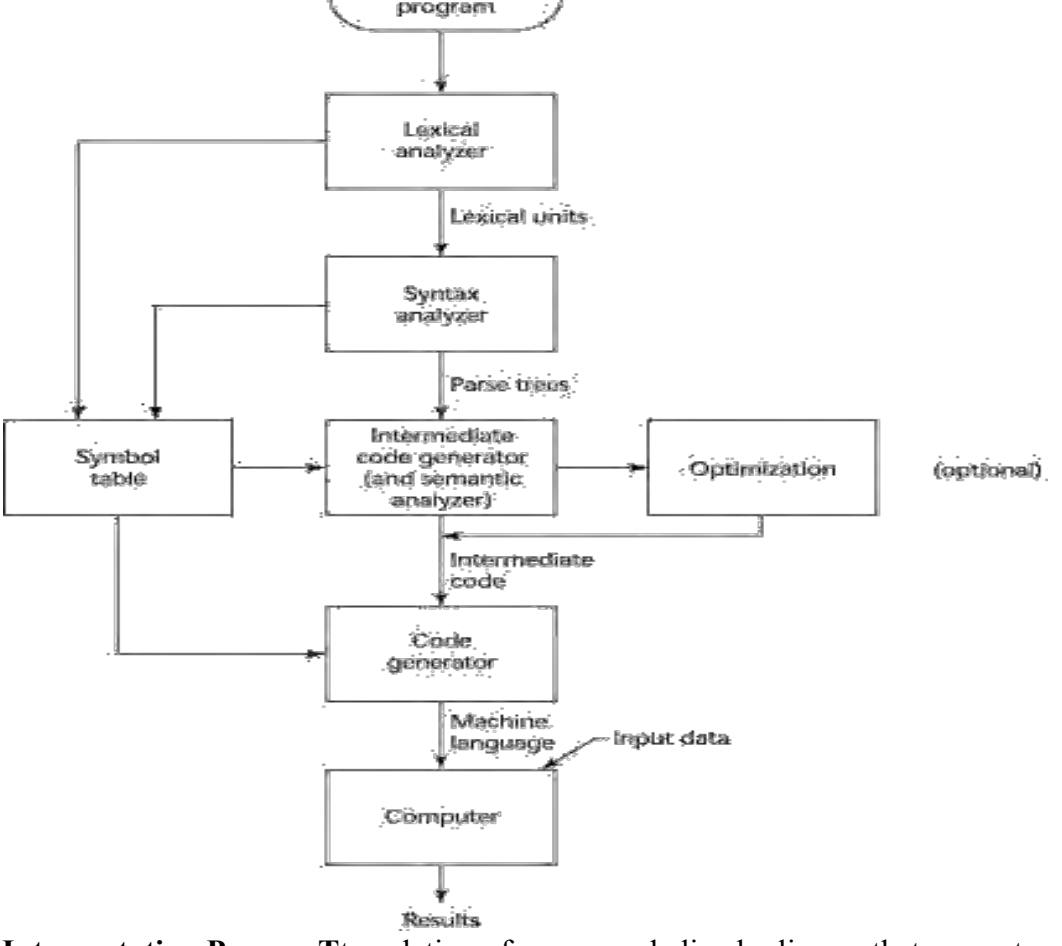
The Compilation Process

Translate high-level program (source language) into machine code (machine language)

• Slow translation, but fast execution.

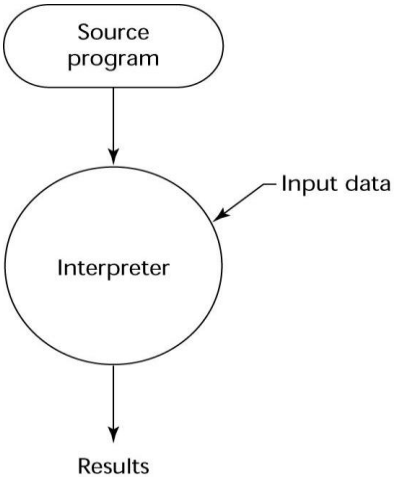
Translates whole source code into object code at a time and generate errors at the end . If no errors ,generates object code. The object code after linking with libraries generates exe code. User input will be given with execution.

Ex: C++ is a compiler



Pure Interpretation Process Ttranslation of source code line by line so that generates errors line by line.If no errors in the code generates object code. While interpretation it selef user input will be given.

- Immediate feedback about errors
- Slower execution
- Often requires more space



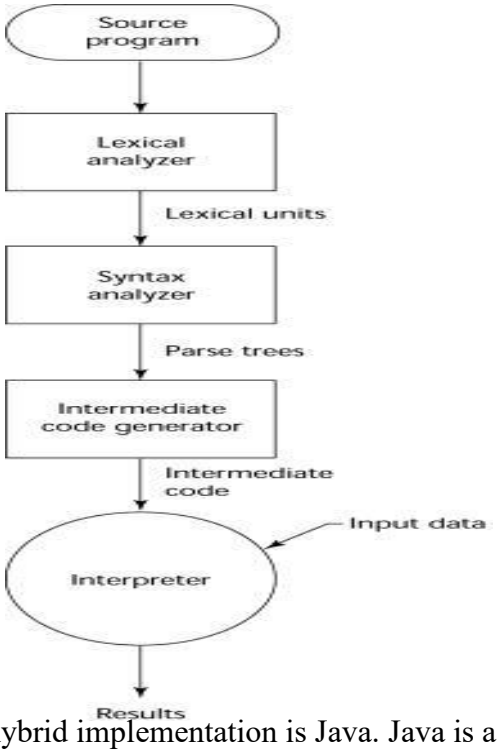
- Used mainly for scripting languages

Example for interpreter is Dbase III plus

Hybrid Implementation Process

A compromise between compilers and pure interpreters

- A high-level language program is is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation



Example for hybrid implementation is Java. Java is a compiled interpreted language.

Just-in-Time Implementation Systems

•Initially translate programs to an intermediate language

• Then compile intermediate language into machine code

• Machine code version is kept for subsequent calls

• JIT systems are widely used for Java programs

• .NET languages are implemented with a JIT system

`

Preprocessors

• Pre-processor macros (instructions) are commonly used to specify that code from another file is to be included

• A pre-processor processes a program immediately before the program is compiled to expand
embedded pre-processor macros

• A well-known example: C pre-processor
  – expands #include, #define, and similar macros

Programming Environments'

- The collection of tools used in software development
- The old way used the console and independent tools
  - UNIX/Linux
- vi or emacs for editing
- compiler
- debugger
- Integrated Development Environments provide a graphical interface to most of the necessary tools

Integrated Development Environments

- Eclipse
  - An integrated development environment for Java, written in java
  - Support for other languages is also available
- Borland JBuilder, NetBeans
  - Other integrated development environments for Java
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

UNIT 2

Describing Syntax and Semantics

- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
- Pragmatics:
- Syntax and semantics provide a language's definition
  - Users of a language definition
    Other language designers
    Implementers

Programmers (the users of the language)

Terminology

- A metalanguage is a language used to describe another language
- A sentence is a string of characters over some alphabet
- A language is a set of sentences
  - a language is specified by a set of rules
  - A lexeme is the lowest level syntactic unit of a language (e.g., *, sum, begin)
  - A token is a category of lexemes (e.g., identifier)

Two approaches to Language Definition

- Recognizers
  - Read a string and decide whether it follows the rules for the language
  - Example: syntax analysis part of a compiler
- Generators
  - A device that generates sentences of a language (BNF)
  - More useful for specifying the language than for checking a string

Syntax

- Backus-Naur Form and Context-Free Grammars
  - Most widely known method for describing programming language syntax
  - Developed as part of the process for specifying ALGOL
  - Define a class of languages called context-free languages
- Extended BNF
  - Improves readability and writability of BNF

BNF Fundamentals

Non-terminals: BNF abstractions used to represent classes of syntactic structures

Terminals: lexemes and tokens
Grammar: a collection of rules

Examples of BNF rules:
<ident_list> → identifier
| identifier, <ident_list>
<if_stmt> → **if** <logic_expr> **then** <stmt>

BNF Rules

A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols

In a context-free grammar, there can only be one symbol on the LHS

A grammar is a finite nonempty set of rules

An abstraction (or nonterminal symbol) can have more than one RHS

Derivations

- BNF is a generative device
  - Use a grammar to generate sentences that belong to the language the grammar describes
- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Derivation

- Every string of symbols in the derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

An Example Grammar

<program> -> <stmts>

<stmts> -> <stmt> | <stmt> ; <stmts> <stmt> -> <var> = <expr>

<var> -> a | b | c | d

<expr> -> <term> + <term> | <term> - <term>

<term> -> <var> | const

An Example Derivation

<program> => <stmts> => <stmt> => <var> = <expr> => a = <expr>

=> a = <term> + <term> => a = <var> + <term> => a = b + <term>

=> a = b + const

Parse Tree

A hierarchical representation of a DERIVATION

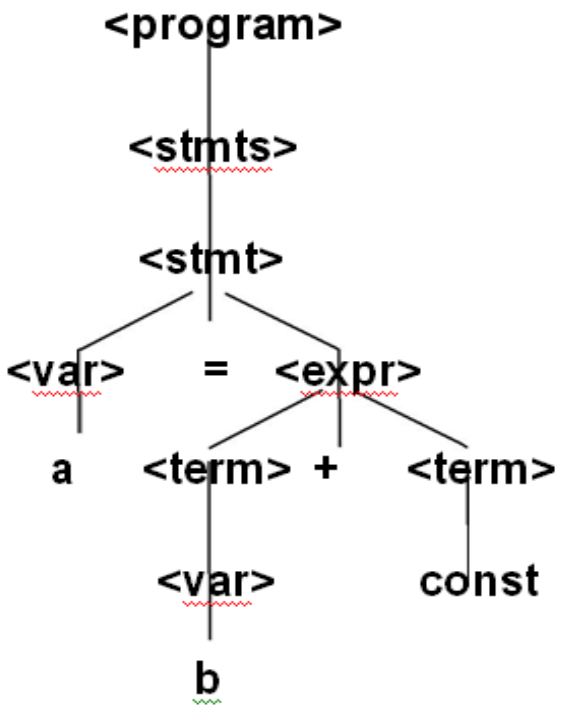
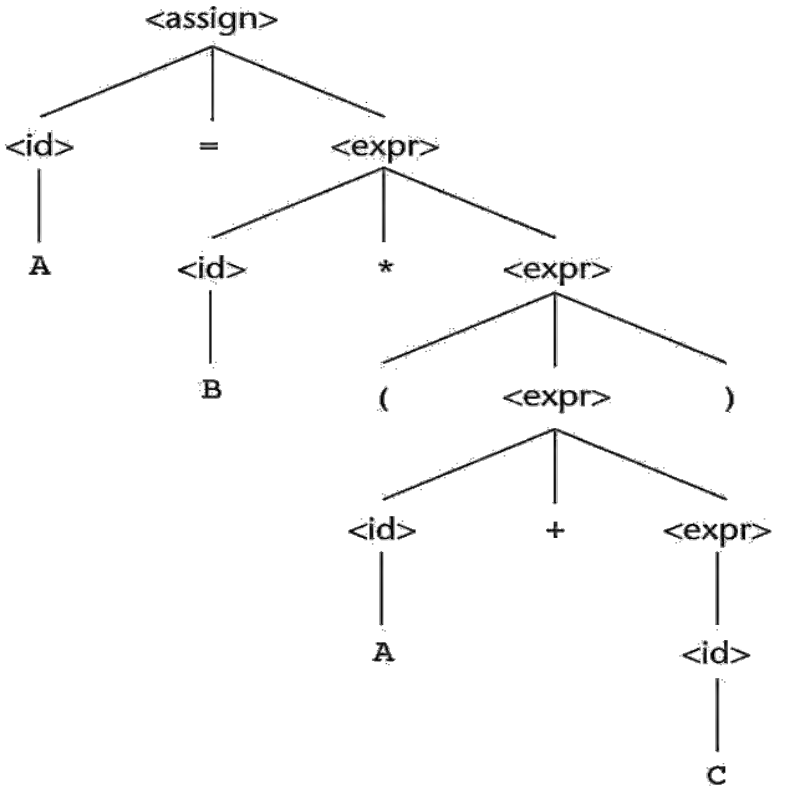


Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`



Associativity of Operators

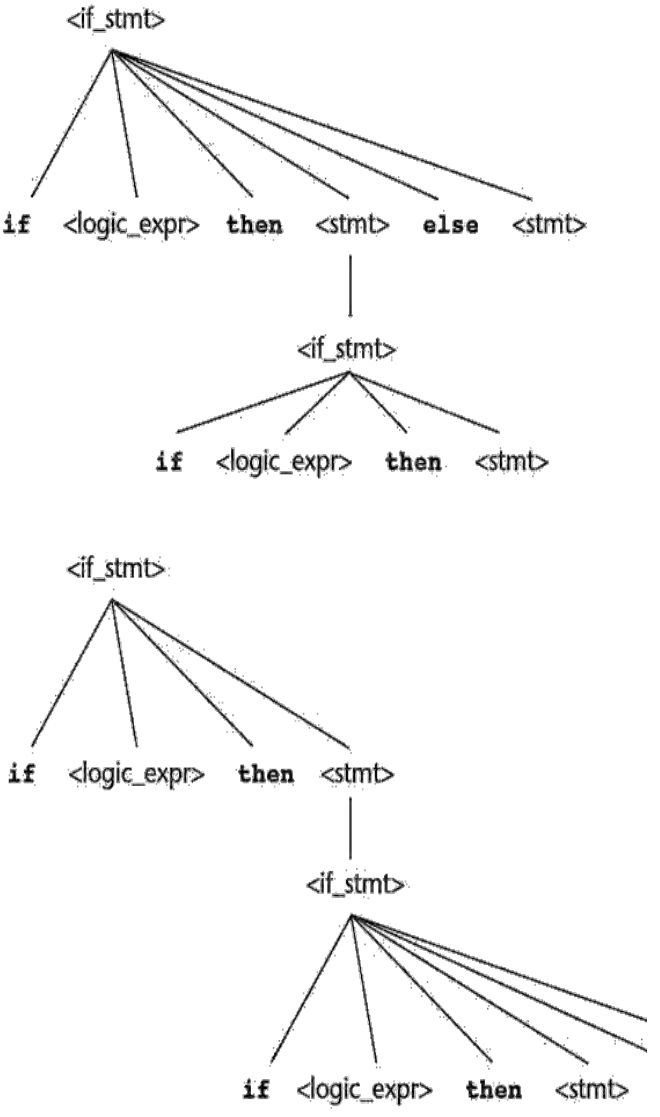
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- For example

  – the requirement that a variable be declared before it can be used is impossible to express in a grammar

– information about variable and expression types could be included in a grammar but only at the cost of great complexity

Figure 3.5

Two distinct parse trees for the same sentenial form



Names and its Design issues.

Imperative languages are abstractions of von Neumann architecture

- A machine consists of
  - Memory - stores both data and instructions
  - Processor - can modify the contents of memory
- Variables are used as an abstraction for memory cells
  - For primitive types correspondence is direct
  - For structured types (objects, arrays) things are more complicated

Names

- Why do we need names?
- need a way to refer to variables, functions, user-defined types, labeled statements,
  Design issues for names:
  - Maximum length?
  - What characters are allowed?
  - Are names case sensitive?
    - C, C++, and Java names are case sensitive
    - this is not true of other languages
  - Are special words reserved words or keywords?

Length of Names

- If too short, they cannot be connotative
- Language examples:
  - FORTRAN I: maximum 6
  - COBOL: maximum 30
  - FORTRAN 90 and ANSI C: maximum 31
  - Ada and Java: no limit, and all are significant
  - C++: no limit, but implementers often impose one

Keywords vs Reserved Words

- Words that have a special meaning in the language
- A *keyword* is a word that is special only in certain contexts, e.g., in Fortran

  \endash Real VarName (Real *is a data type followed with a name, therefore* Real *is a keyword)*

  \endash Real = 3.4 (*Real is a variable)*

- A *reserved word* is a special word that cannot be used as a user-defined name
  - most reserved words are also keywords

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variable Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated (l-value)
- *Type* - allowed range of values of variables and the set of defined operations
- *Value* - the contents of the location with which the variable is associated (r-value)

The concept of Binding and possible Binding times.

The Concept of Binding

- A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol
  - entity could be a variable or a function or even a class
- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a FORTRAN 77 variable to a memory cell (or a C static variable)
- Runtime -- bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.

- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Type Binding

\endash How is a type specified?

  - An *explicit declaration* is a program statement used for declaring the types of variables
  - An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)

\endash When does the binding take place?

\endash If static, the type may be specified by either an explicit or an implicit declaration

Type checking and how coercion related to type checking?

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
- A *type error* is the application of an operator to an operand of an inappropriate type

When is type checking done?

- If all type bindings are static, nearly all type checking can be static (done at compile time)
- If type bindings are dynamic, type checking must be dynamic (done at run time)
- A programming language is *strongly typed* if type errors are always detected
  - Advantage: allows the detection of misuse of variables that result in type errors

How strongly typed?

  - FORTRAN 77 is not: parameters, EQUIVALENCE
  - Pascal has variant records
  - C and C++
    - parameter type checking can be avoided
    - unions are not type checked

- Ada is almost
    - UNCHECKED CONVERSION is loophole
- Java is similar

Coercion

- The automatic conversion between types is called coercion.
- Coercion rules they can weaken typing considerably
    - C and C++ allow both widening and narrowing coercions
    - Java allows only widening coercions
        - Java's strong typing is still far less effective than that of Ada

Dynamic type binding

- Dynamic Type Binding (Perl, JavaScript and PHP, Scheme)
- Specified through an assignment statement e.g., JavaScript

```
list = [2, 4.33, 6, 8];

list = 17.3;
```

- This provides a lot of flexibility

Storage Bindings & Lifetime

- The lifetime of a variable is the time during which it is bound to a particular memory cell
    - Allocation - getting a cell from some pool of available cells
    - Deallocation - putting a cell back into the pool
- Depending on the language, allocation can be either controlled by the programmer or done automatically

Categories of Variables by Lifetimes

- Static--lifetime is same as that of the program
- Stack-dynamic--lifetime is duration of subprogram
- *Explicit heap-dynamic* -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements

Variable Scope

- The *scope* of a variable is the range of statements over which it is visible

- The *nonlocal variables* of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables
- Scope and lifetime are sometimes closely related, but are different concepts

Static Scope

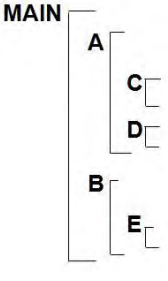
- The scope of a variable can be determined from the program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent

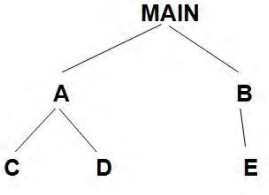
Scope and Shadowed Variables

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++, Java and Ada allow access to some of these "hidden" variables
  - In Ada: **unit.name**
  - In C++: **class_name::name** or **::name** for globals
  - In Java: **this.name** for variables declared at class level

Static Scoping and Nested Functions

- Assume MAIN calls A and B
  A calls C and D
  B calls A and E





Nested Functions and Maintainability

- Suppose the spec is changed so that D must now access some data in B
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Using nested functions with static scoping often encourages many globals
  - not considered to be good programming practice
  - Current thinking is that we can accomplish the same thing better with modules (classes)
- Static scoping is still preferred to the alternative
  - Most current languages use static scoping at the block level even if they don't allow nested functions

Object Lifetime and Storage Management

The period of time between the creation and the destruction of a name-toobject binding is called the binding's *lifetime*. Similarly, the time between the creation and destruction of an object is the object's lifetime.

an object may retain its value and the potential to be accessed even when a given name can no longer be used to access it.

When a variable is passed to a subroutine by *reference*, for example (as it typically is in Fortran or with var parameters in Pascal or "&" parameters in C++), the binding between the parameter name and the variable that was passed has a lifetime shorter than that of the variable itself.

A binding to an object that is no longer live is called a *dangling reference*.

Object lifetimes generally correspond to one of three principal *storage allocation* mechanisms, used to manage the object's space:

- *Static* objects are given an absolute address that is retained throughout the program's execution.
- *Stack* objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
- *Heap* objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm.

Static Allocation

Global variables are the obvious example of static objects, but not the only one.The instructions that constitute a program's machine-language translation can also be thought of as statically allocated objects.

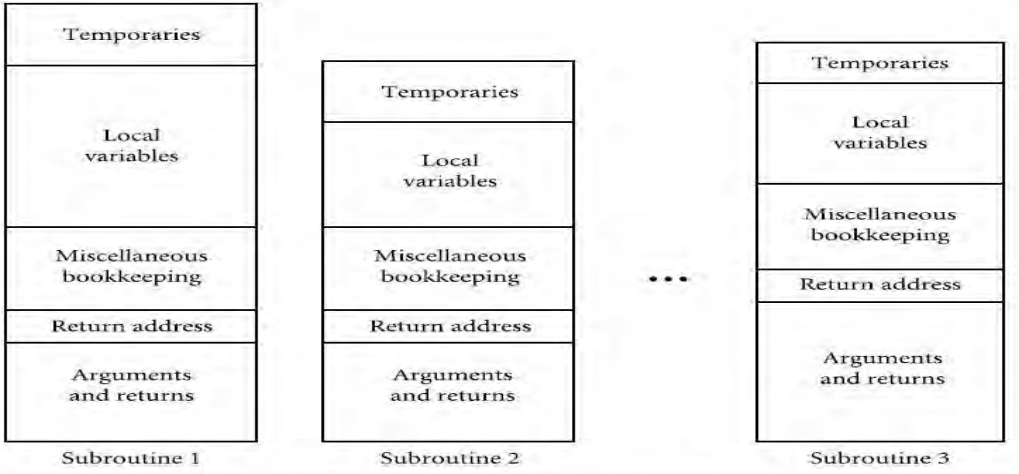
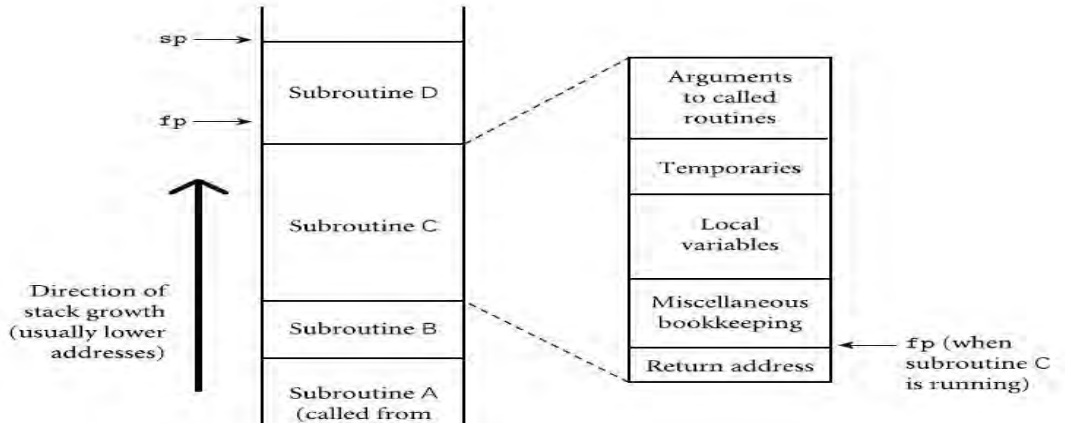


Figure 3.1 Static allocation of space for subroutines in a language or program without recursion.

Logically speaking, local variables are created when their subroutine is called and destroyed when it returns. If the subroutine is called repeatedly, each invocation is said to create and destroy a separate *instance* of each local variable.

Most compilers produce a variety of tables that are used by runtime support routines for debugging, dynamic type checking, garbage collection, exception handling, and other purposes; these are also statically allocated.
Recursion was not originally supported in Fortran (it was added in Fortran 90). As a result, there can never be more than one invocation of a subroutine active at any given time, and a compiler may choose to use static allocation for local variables, effectively arranging for the variables of different invocations to share the same locations, and thereby avoiding any run-time overhead for creation and destruction as shown in the above figure3.1.

Along with local variables and elaboration-time constants, the compiler typically stores a variety of other information associated with the subroutine, including the following.
Arguments and return values. Modern compilers tend to keep these in registers when possible, but sometimes space in memory is needed.
Temporaries. These are usually intermediate values produced in complex calculations. Again, a good compiler will keep them in registers whenever possible.
Bookkeeping information. This may include the subroutine's return address, a reference to the stack frame of the caller (also called the *dynamic link*), additional saved registers, debugging information.
Stack-Based Allocation: If a language permits recursion, static allocation of local variables is no longer an option, since the number of instances of a variable that may need to exist at the same time is conceptually unbounded. Fortunately, the natural nesting of subroutine calls makes it easy to allocate space for locals on a stack.



In the stack babsed allocation Each instance of a subroutine at run time has its own *frame* (also called an *activation record*) on the stack, containing arguments and return values, local variables, temporaries, and bookkeeping information. Arguments to be passed to subsequent routines lie at the top of the frame, where the callee can easily find them. The organization of the remaining information is implementation-dependent: it varies from one language and compiler to another.

Maintenance of the stack is the responsibility of the subroutine calling sequence—the code executed by the caller immediately before and after the call—and of the prologue (code executed at the beginning) and epilogue (code executed at the end) of the subroutine itself. Sometimes the term"calling sequence" is used to refer to the combined operations of the caller, the prologue, and the epilogue.

Heap-Based Allocation

A heap is a region of storage in which subblocks can be allocated and deallocated at arbitrary times. Heaps are required for the dynamically allocated pieces of linked data structures and for dynamically resized objects.

There are many possible strategies to manage space in a heap. The principal concerns are speed and space, and as usual there are tradeoffs between them. Space concerns can be further subdivided into issues of internal and external *fragmentation*. Internal fragmentation occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object; the extra space is then unused.

External fragmentation occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks: there may be quite a lot of free space, but no one piece of it may be large enough to satisfy some future request.

Initially the list consists of a single block comprising the entire heap. At each allocation request the algorithm searches the list for a block of appropriate size.

With a first fit algorithm we select the first block on the list that is large enough to satisfy the request.

With a best fit algorithm we search the entire list to find the smallest block that is large enough to satisfy the request. In either case, if the chosen block is significantly larger than required, then we divide it in two and return the unneeded portion to the free list as a smaller block.

In any algorithm that maintains a single free list, the cost of allocation is linear in the number of free blocks.

To reduce this cost to a constant, some storage management algorithms maintain separate free lists for blocks of different sizes.

Each request is rounded up to the next standard size (at the cost of internal fragmentation) and allocated from the appropriate list. In effect, the heap is divided into "pools," one for each standard size. The division may be static or dynamic.

Two common mechanisms for dynamic pool adjustment are known as the **buddy system and the Fibonacci heap.**

In the **buddy system**, the standard block sizes are powers of two. If a block of size 2k is needed, but none is available, a block of size 2k+1 is split in two.

One of the halves is used to satisfy the request; the other is placed on the kth free list. When a block is deallocated, it is coalesced with its "buddy"—the other half of the split that created it—if that buddy is free.

Fibonacci heaps are similar, but they use Fibonacci numbers for the standard sizes, instead of powers of two. The algorithm is slightly more complex but leads to slightly lower

internal fragmentation because the Fibonacci sequence grows more slowly than 2k.

Garbage Collection in storagemanagement

Allocation of heap-based objects is always triggered by some specific operation in a program: instantiating an object, appending to the end of a list, assigning a long value into a previously short string, and so on. Deallocation is also explicit in some languages (e.g., C, C++, and Pascal.).delete operator in C++ will destroy the object as free() function in C destroy the memory allocated.

However, Some languages specify that objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable. In Java automatic garbage collector will call finalize() to destroy the object. The run-time library for such a language

must then provide a garbage collection mechanism to identify and reclaim unreachable objects. Most functional languages require garbage collection, as do many more recent imperative languages, includingModula-3, Java,C#, and all the major scripting languages.

However, is compelling:manual deallocation errors are among the most common and costly bugs in real-world programs. If an object is deallocated too soon, the program may follow a ***dangling reference***, accessing memory now used by another object.

If an object is *not* deallocated at the end of its lifetime, then the program may "leak memory," eventually running out of heap space. Deallocation errors are notoriously difficult to identify and fix.

Over time, both language designers and programmers have increasingly come to consider automatic garbage collection an essential language feature.

Garbage-collection algorithms have improved, reducing their run-time overhead; language implementations have become more complex in general, reducing the marginal complexity of automatic collection.

Module, Module type and Module manager

A major challenge in the construction of any large body of software is how to divide the effort among programmers in such a way that work can proceed on multiple fronts simultaneously.

This modularization of effort depends critically on the notion of *information hiding*, which makes objects and algorithms invisible,whenever possible, to portions of the systemthat do not need them.

In a well-designed program the interfaces between modules are as "narrow" (i.e., simple) as possible, and any design decision that is likely to change is hidden inside a single module.

A module allows a collection of objects—subroutines, variables, types, and so on—to be encapsulated in such a way that (1) objects inside are visible to each other, but (2) objects on the inside are not visible on the outside unless explicitly exported, and (3) objects outside are not visible on the inside unless explicitly imported.

Modules can be found in Clu (*clusters*), Modula (1, 2, and 3), Turing, Ada, Java (*packages*),C++ (*namespaces*), and many other modern languages.

Most module-based languages allow the programmer to specify that certain exported names are usable only in restricted ways.

Modules into which names must be explicitly imported are said to be *closed scopes*. Modules are closed in Modula (1, 2, and 3). By extension, modules that do not require imports are said to be *open scopes*.

Nested subroutines are open scopes in most Algol family languages. Important exceptions are Euclid, in which both module and subroutine scopes are closed scoped.

Module Types and Classes

Modules facilitate the construction of abstractions by allowing data to be made private to the subroutines that use them. A module type, through which the programmer can declare an arbitrary number of similar module objects.

The skeleton of a Euclid stack appears in Figure 3.9 specify finalization code that will be executed at the end of a module's lifetime.

```
CONST stack_size = ...
TYPE element = ...
...
MODULE stack_manager;
IMPORT element, stack_size;
EXPORT stack, init_stack, push, pop;
TYPE
    stack_index = [1..stack_size];
    STACK = RECORD
        s : ARRAY stack_index OF element;
        top : stack_index;          (* first unused slot *)
    END;

PROCEDURE init_stack(VAR stk : stack);
BEGIN
    stk.top := 1;
END init_stack;

PROCEDURE push(VAR stk : stack; elem : element);
BEGIN
    IF stk.top = stack_size THEN
        error;
    ELSE
        stk.s[stk.top] := elem;
        stk.top := stk.top + 1;
    END;
END push;

PROCEDURE pop(VAR stk : stack) : element;
BEGIN
    IF stk.top = 1 THEN
        error;
    ELSE
        stk.top := stk.top - 1;
        return stk.s[stk.top];
    END;
END pop;

END stack_manager;
```

```
var A, B : stack;
var x, y : element;
...
init_stack(A);
init_stack(B);
...
push(A, x);
...
y := pop(B);
```

Figure 3.8 Manager module for stacks in Modula-2.

```
const stack_size := ...
type element : ...
...
type stack = module
    imports (element, stack_size)
    exports (push, pop)
type
    stack_index = 1..stack_size
var
    s   : array stack_index of element
    top : stack_index

procedure push(elem : element) = ...
function pop returns element = ...
...
initially
    top := 1
end stack
```

```
var A, B : stack
var x, y : element
...
A.push(x)
...
y := B.pop
```

Figure 3.9 Module type for stacks in Euclid. Unlike the code in Figure 3.7, the code here can

The difference between the module-as-manager and module-as-type approaches to abstraction is reflected in the lower right of Figures 3.8 and 3.9. With

module types, the programmer can think of the module's subroutines as "belonging" to the stack in question (A.push(x)), rather than as outside entities to which the stack can be passed as an argument (push(A, x)).

UNIT –III

Semantic analysis

Semantic analysis and intermediate code generation can be described in terms of annotations/decorations of a parse tree.

.Annotations are themselves are known as attributes.

Attribute grammar provides a framework for the decoration of a tree.

This framework is a useful conceptual tool even in compliers that do not build a parse tree as an explicit data structures.

Dynamic checks:

In the program development and testing dynamic checks are followed. Some languages will make dynamic checks as a part of their complier. some compilers even after including it is a part , as and when necessary it can be disabled to improve the performance.

Assertions in Euclid ,Effel allow programmers to specify logical assertions, alert, invariants, pre conditions and post conditions that must be verified by dynamic semantic checks.

An assertion is a statement that a specific condition is expected to be true when execution reaches a certain point in the code.

Ex: in Euclid asseret can be

assert den not=0 where den is denominator is integer variable. In 'Ç' assert can be **assert(den!=0);**

in C++ C type assert is allowed and also if necessary you can use cerror<<assertion";

In Java Exception handler mechanism can achive same effect of assertion.

In Oracle Developer 2000 alert is similar to assert in C.

ALERTS in D2K

step 1 : Go to alert

step 2 : create alert name ---------->using f4

step 3 : Select DEFALT_ALERT_BUTTON ---------- > Give {LIST ITEM } button

step 4 : create any auther function button ---------->Eg SAVE Button

step 5 : Write a qury in that function button ---------- > Go to {WHEN_BUTTON_PRESED}

==========

SAMPLE CODING:=

==========

```
declare
a_ _number
begin
a:=show_alert('in_save');
if_ _a=alert_button1 then

commit_form;
end if;
end;
```

Static analysis:

Compile time algorithms that predict runtime behaviour is referred as static analysis.

Ex: Type checking in Ada,C, ML

The compiler will ensures that no variable will ever be used at runtime in a way that is in appropriate. For its type.

So these languages are not type safe.

But some languages like LISP, Small talk are type safe by accepting the run time behaviour overhead of dynamic type checks.

Optimization:

If we think of omission of unnecessary dynamic checks as performance optimization ,it is natural to look for other ways in which static analysis may enable code improvement.

An optimization is said to be unsafe if it may leads to incorrect code in certain programs. It is said to be Speculative optimization.

Speculative optimization usually improves performance but may degrade it in certin cases.

A compiler is said to be **conservative** if it applies optimization only when it can guarantee that they will be both safe and effective.

Ex: In speculative optimization it includes non binding prefetches, which try to bind data in to cache before they are needed and trace scheduling , which arranges code in hopes of improving the performance of the processor pipeline and the instruction cache.

Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis
  - (intermediate) code generation
- The principal job of the semantic analyzer is to enforce static semantic rules
  - constructs a syntax tree (usually first)
  - information gathered is needed by the code generator
- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved
- A common approach interleaves construction of a syntax tree with parsing (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation
- The PL/0 compiler has no optimization to speak of (there's a tiny little trivial phase, which operates on the syntax tree)

- Its code generator produces MIPs assembler, rather than a machine-independent intermediate form

Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree
- ATTRIBUTE GRAMMARS provide a formal framework for decorating such a tree
- The notes below discuss attribute grammars and their ad-hoc cousins, ACTION ROUTINES
- We'll start with decoration of parse trees, then consider syntax trees
- Consider the following LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

  $E \rightarrow E + T$
  $E \rightarrow E - T$
  $E \rightarrow T$
  $T \rightarrow T * F$
  $T \rightarrow T / F$
  $T \rightarrow F$
  $F \rightarrow - F$
  $F \rightarrow (E)$
  $F \rightarrow const$

- This says nothing about what the program MEANS
- We can turn this into an attribute grammar as follows (similar to Figure 4.1):

$E \rightarrow E + T$	E1.val = E2.val + T.val
$E \rightarrow E - T$	E1.val = E2.val - T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T1.val = T2.val * F.val
$T \rightarrow T / F$	T1.val = T2.val / F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow - F$	F1.val = - F2.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow const$	F.val = C.val

- The attribute grammar serves to define the semantics of the input program
- Attribute rules are best thought of as definitions, not assignments
- They are not necessarily meant to be evaluated at any particular time, or in any particular order, though they do define their left-hand side in terms of the right-hand side

Evaluating Attributes

- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree [see Figure 4.2 for (1+3)*2]
  - When a parse tree under this grammar is fully decorated, the value of the expression will be in the *val* attribute of the root

- The code fragments for the rules are called SEMANTIC FUNCTIONS
  - Strictly speaking, they should be cast as functions, e.g., E1.val = sum (E2.val, T.val), cf., Figure 4.1

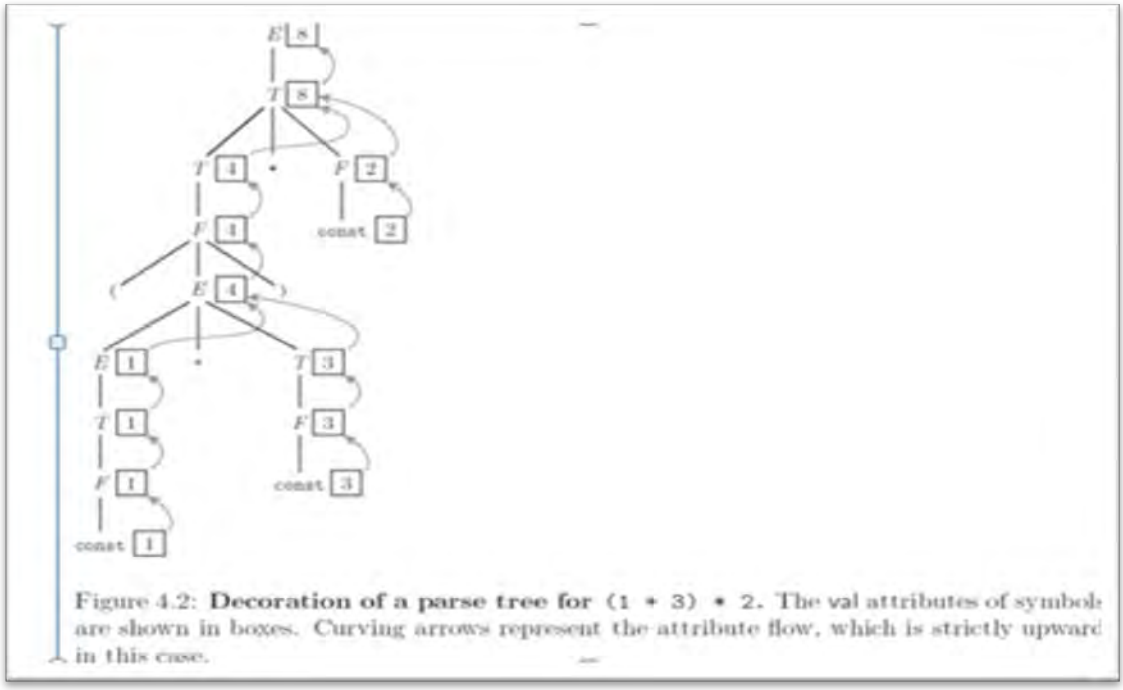


Figure 4.2: **Decoration of a parse tree for** `(1 * 3) * 2`. The val attributes of symbols are shown in boxes. Curving arrows represent the attribute flow, which is strictly upward in this case.

- This is a very simple attribute grammar:
  - Each symbol has at most one attribute
    - the punctuation marks have no attributes
- These attributes are all so-called SYNTHESIZED attributes:
  - They are calculated only from the attributes of things below them in the parse tree
- In general, we are allowed both synthesized and INHERITED attributes:
  - Inherited attributes may depend on things above or to the side of them in the parse tree
  - Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).
  - Inherited attributes of the start symbol constitute run-time parameters of the compiler
- The grammar above is called S-ATTRIBUTED because it uses only synthesized attributes
- Its ATTRIBUTE FLOW (attribute dependence graph) is purely bottom-up
  - It is SLR(1), but not LL(1)
- An equivalent LL(1) grammar requires inherited attributes:

Evaluating Attributes – Example

Attribute grammar

E	→ T TT	E.v = TT.v
		TT.st = T.v
TT1	→ + T TT2	TT1.v = TT2.v
		TT2.st = TT1.st + T.v
TT1	→ - T TT1	TT1.v = TT2.v
		TT2.st = TT1.st - T.v
TT	→ ε	TT.v = TT.st
T	→ F FT	T.v = FT.v
		FT.st = F.v

- Attribute grammar

FT1	→ * F FT2	FT1.v = FT2.v
		FT2.st = FT1.st * F.v
FT1	→ / F FT2	FT1.v = FT2.v
		FT2.st = FT1.st / F.v
FT	→ ε	FT.v = FT.st
F1	→ - F2	F1.v = - F2.v
F	→ (E)	F.v = E.v
F	→ const	F.v = C.v

- Figure 4.4 – parse tree for (1+3)*2

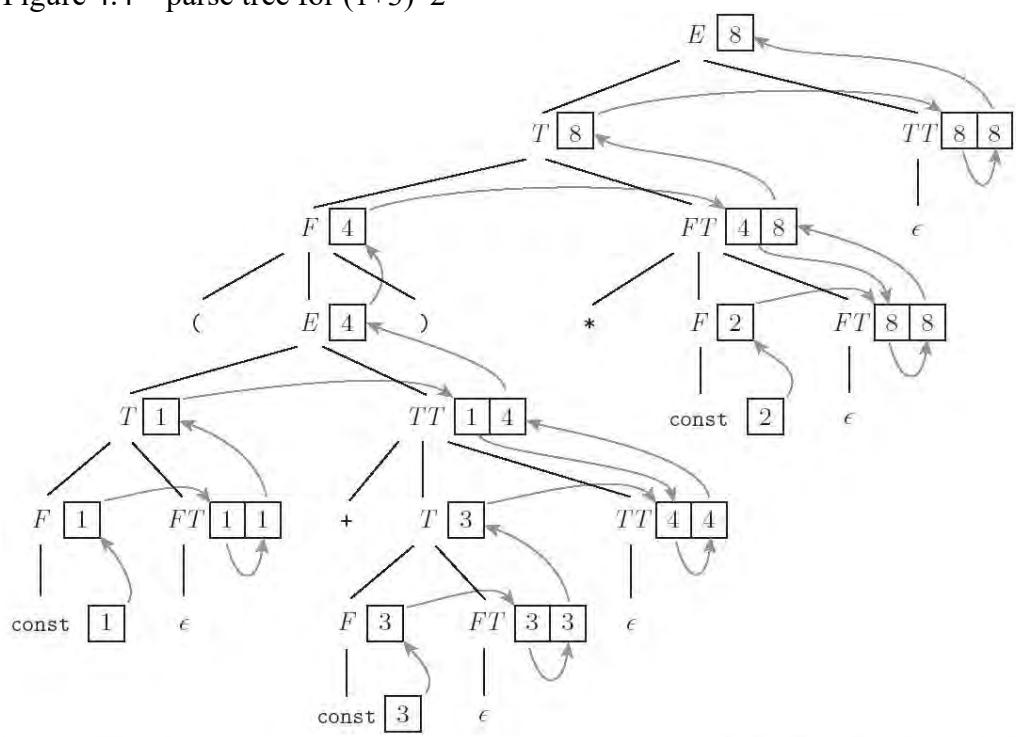


Figure 4.4: **Decoration of a top-down parse tree for `(1 + 3) * 2`, using the attribute grammar of Figure 4.3.** Curving arrows again represent attribute flow, which is no longer bottom-up, but is still left-to-right.

- This attribute grammar is a good bit messier than the first one, but it is still L-ATTRIBUTED, which means that the attributes can be evaluated in a single left-to-right pass over the input
- In fact, they can be evaluated during an LL parse
- Each synthetic attribute of a LHS symbol (by definition of synthetic) depends only on attributes of its RHS symbols
- Each inherited attribute of a RHS symbol (by definition of L-attributed) depends only on
    - inherited attributes of the LHS symbol, or
    - synthetic or inherited attributes of symbols to its left in the RHS
- L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse

Evaluating Attributes

- There are certain tasks, such as generation of code for short-circuit Boolean expression evaluation, that are easiest to express with non-L-attributed attribute grammars
- Because of the potential cost of complex traversal schemes, however, most real-world compilers insist that the grammar be L-attributed

$E_1 \longrightarrow E_2$ + T
    ▷ $\mathsf{E}_1.\mathsf{ptr} := \mathsf{make\_bin\_op}(\text{“+”}, \mathsf{E}_2.\mathsf{ptr}, \mathsf{T.ptr})$

$E_1 \longrightarrow E_2$ - T
    ▷ $\mathsf{E}_1.\mathsf{ptr} := \mathsf{make\_bin\_op}(\text{“}-\text{”}, \mathsf{E}_2.\mathsf{ptr}, \mathsf{T.ptr})$

$E \longrightarrow T$
    ▷ $\mathsf{E.ptr} := \mathsf{T.ptr}$

$T_1 \longrightarrow T_2$ * F
    ▷ $\mathsf{T}_1.\mathsf{ptr} := \mathsf{make\_bin\_op}(\text{“}\times\text{”}, \mathsf{T}_2.\mathsf{ptr}, \mathsf{F.ptr})$

$T_1 \longrightarrow T_2$ / F
    ▷ $\mathsf{T}_1.\mathsf{ptr} := \mathsf{make\_bin\_op}(\text{“}\div\text{”}, \mathsf{T}_2.\mathsf{ptr}, \mathsf{F.ptr})$

$T \longrightarrow F$
    ▷ $\mathsf{T.ptr} := \mathsf{F.ptr}$

$F_1 \longrightarrow$ - F_2
    ▷ $\mathsf{F}_1.\mathsf{ptr} := \mathsf{make\_un\_op}(\text{“}^{+}/_{-}\text{”}, \mathsf{F}_2.\mathsf{ptr})$

$F \longrightarrow$ (E)
    ▷ $\mathsf{F.ptr} := \mathsf{E.ptr}$

$F \longrightarrow$ `const`
    ▷ $\mathsf{F.ptr} := \mathsf{make\_leaf}(\mathsf{const.val})$

Figure 4.5: **Bottom-up attribute grammar to construct a syntax tree.** The symbol $^{+}/_{-}$ is used (as it is on calculators) to indicate change of sign.

$E \longrightarrow T\ TT$
- ▷ TT.st := T.ptr
- ▷ E.ptr := TT.ptr

$TT_1 \longrightarrow$ + $T\ TT_2$
- ▷ TT_2.st := make_bin_op("+", TT_1.st, T.ptr)
- ▷ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow$ - $T\ TT_2$
- ▷ TT_2.st := make_bin_op("−", TT_1.st, T.ptr)
- ▷ TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$
- ▷ TT.ptr := TT.st

$T \longrightarrow F\ FT$
- ▷ FT.st := F.ptr
- ▷ T.ptr := FT.ptr

$FT_1 \longrightarrow$ * $F\ FT_2$
- ▷ FT_2.st := make_bin_op("×", FT_1.st, F.ptr)
- ▷ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow$ / $F\ FT_2$
- ▷ FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)
- ▷ FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$
- ▷ FT.ptr := FT.st

$F_1 \longrightarrow$ - F_2
- ▷ F_1.ptr := make_un_op("+/−", F_2.ptr)

$F \longrightarrow$ (E)
- ▷ F.ptr := E.ptr

$F \longrightarrow$ `const`
- ▷ F.ptr := make_leaf(const.val)

Figure 4.6: **Top-down attribute grammar to construct a syntax tree.** Here the st attribute, like the ptr attribute (and unlike the st attribute of Figure 4.3), is a pointer to a syntax tree node.

Evaluating Attributes – Syntax Trees

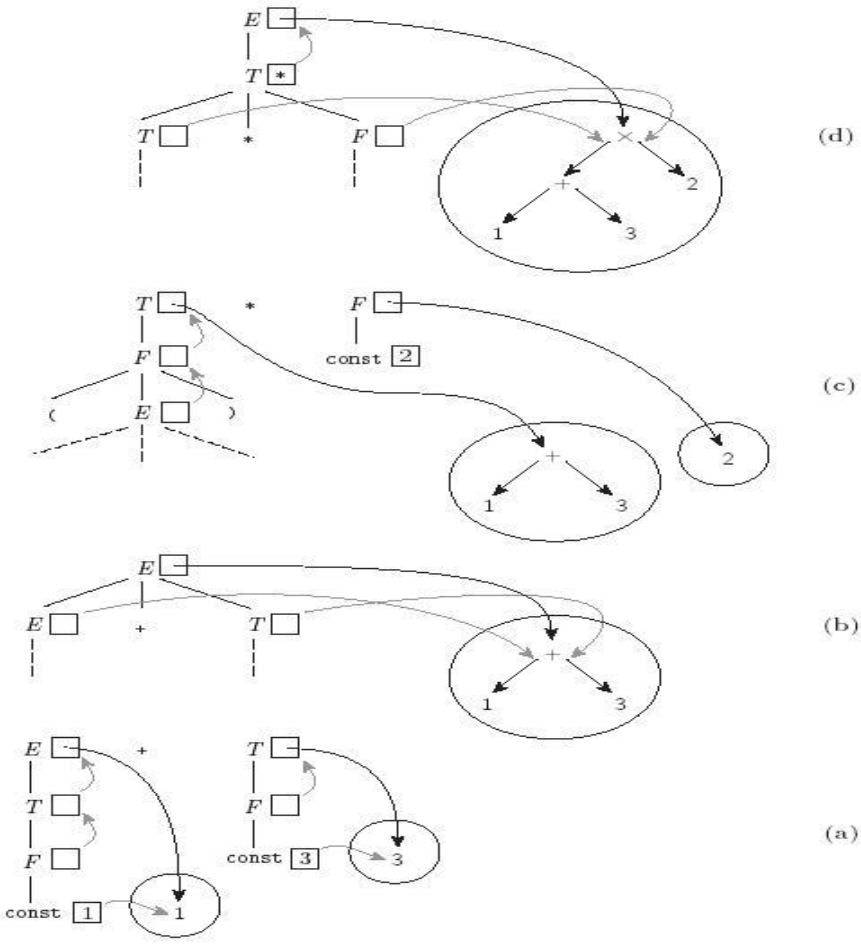


Figure 4.7: **Construction of a syntax tree via decoration of a bottom-up parse tree, using the grammar of Figure 4.5.** In diagram (a), the values of the constants 1 and 3 have been placed in new syntax tree leaves. Pointers to these leaves propagate up into the attributes of E and T. In (b), the pointers to these leaves become child pointers of a new internal + node. In (c) the pointer to this node propagates up into the attributes of T, and a new leaf is created for 2. Finally, in (d), the pointers from T and F become child pointers of a new internal $\times$ node, and a pointer to this node propagates up into the attributes of E.

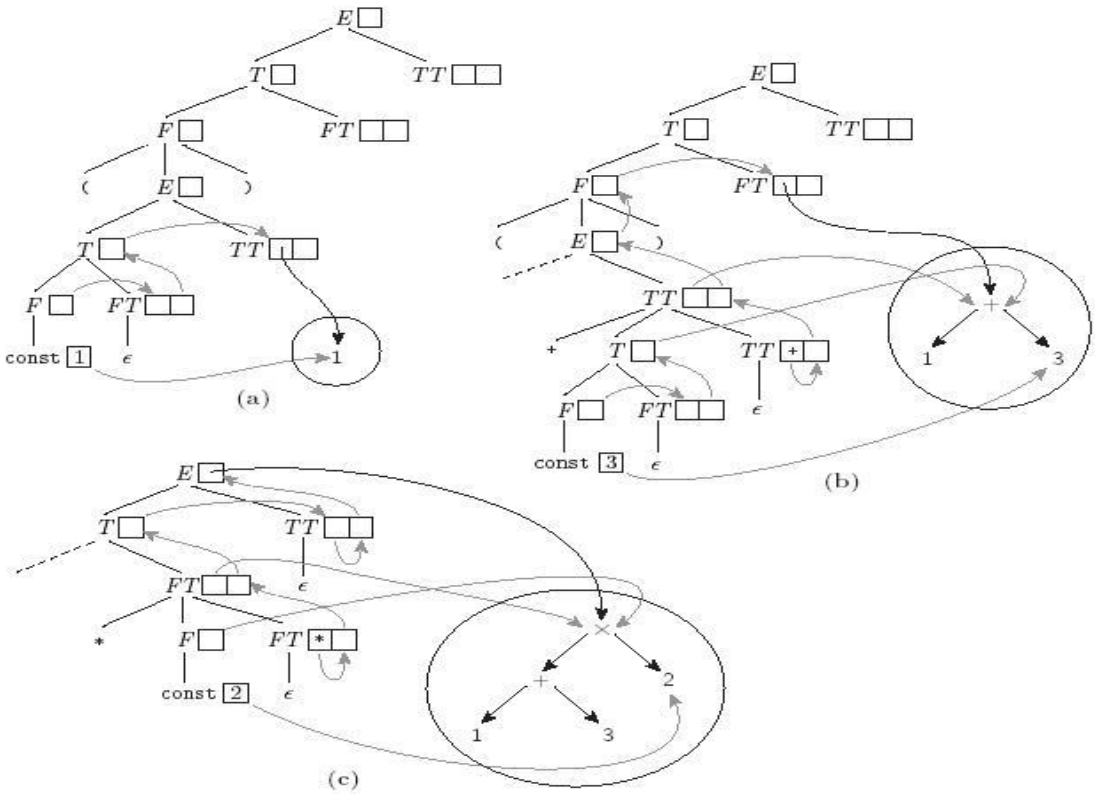


Figure 4.8: **Construction of a syntax tree via decoration of a top-down parse tree, using the grammar of Figure 4.6.** In the top diagram, (a), the value of the constant `1` has been placed in a new syntax tree leaf. A pointer to this leaf then propagates to the `st` attribute of TT. In (b), a second leaf has been created to hold the constant `3`. Pointers to the two leaves then become child pointers of a new internal `+` node, a pointer to which propagates from the `st` attribute of the bottom-most TT, where it was created, all the way up and over to the `st` attribute of the top-most FT. In (c), a third leaf has been created for the constant `2`. Pointers to this leaf and to the `+` node then become the children of a new $\times$ node, a pointer to which propagates from the `st` of the lower FT, where it was created, all the way to the root of the tree.

Action Routines

- We can tie this discussion back into the earlier issue of separated phases v. on-the-fly semantic analysis and/or code generation
- If semantic analysis and/or code generation are interleaved with parsing, then the TRANSLATION SCHEME we use to evaluate attributes MUST be L-attributed
- If we break semantic analysis and code generation out into separate phase(s), then the code that builds the parse/syntax tree must still use a left-to-right (L-attributed) translation scheme
- However, the later phases are free to use a fancier translation scheme if they want

- There are automatic tools that generate translation schemes for context-free grammars or tree grammars (which describe the possible structure of a syntax tree)
    - These tools are heavily used in syntax-based editors and incremental compilers
    - Most ordinary compilers, however, use ad-hoc techniques
- An ad-hoc translation scheme that is interleaved with parsing takes the form of a set of ACTION ROUTINES:
    - An action routine is a semantic function that we tell the compiler to execute at a particular point in the parse
- If semantic analysis and code generation are interleaved with parsing, then action routines can be used to perform semantic checks and generate code
- If semantic analysis and code generation are broken out as separate phases, then action routines can be used to build a syntax tree
    - A parse tree could be built completely automatically
    - We wouldn't need action routines for that purpose
- Later compilation phases can then consist of ad-hoc tree traversal(s), or can use an automatic tool to generate a translation scheme
    - The PL/0 compiler uses ad-hoc traversals that are almost (but not quite) left-to-right
- For our LL(1) attribute grammar, we could put in explicit action routines as follows:

Action Routines – Example

$E \longrightarrow T \{ \text{TT.st} := \text{T.ptr} \}\ TT \{ \text{E.ptr} := \text{TT.ptr} \}$

$TT_1 \longrightarrow \texttt{+}\ T \{ \text{TT}_2\text{.st} := \text{make\_bin\_op}(\text{"+"}, \text{TT}_1\text{.st}, \text{T.ptr}) \}\ TT_2 \{ \text{TT}_1\text{.ptr} := \text{TT}_2\text{.ptr} \}$

$TT_1 \longrightarrow \texttt{-}\ T \{ \text{TT}_2\text{.st} := \text{make\_bin\_op}(\text{"}-\text{"}, \text{TT}_1\text{.st}, \text{T.ptr}) \}\ TT_2 \{ \text{TT}_1\text{.ptr} := \text{TT}_2\text{.ptr} \}$

$TT \longrightarrow \epsilon \{ \text{TT.ptr} := \text{TT.st} \}$

$T \longrightarrow F \{ \text{FT.st} := \text{F.ptr} \}\ FT \{ \text{T.ptr} := \text{FT.ptr} \}$

$FT_1 \longrightarrow \texttt{*}\ F \{ \text{FT}_2\text{.st} := \text{make\_bin\_op}(\text{"}\times\text{"}, \text{FT}_1\text{.st}, \text{F.ptr}) \}\ FT_2 \{ \text{FT}_1\text{.ptr} := \text{FT}_2\text{.ptr} \}$

$FT_1 \longrightarrow \texttt{/}\ F \{ \text{FT}_2\text{.st} := \text{make\_bin\_op}(\text{"}\div\text{"}, \text{FT}_1\text{.st}, \text{F.ptr}) \}\ FT_2 \{ \text{FT}_1\text{.ptr} := \text{FT}_2\text{.ptr} \}$

$FT \longrightarrow \epsilon \{ \text{FT.ptr} := \text{FT.st} \}$

$F_1 \longrightarrow \texttt{-}\ F_2 \{ \text{F}_1\text{.ptr} := \text{make\_un\_op}(\text{"}^+/_-\text{"}, \text{F}_2\text{.ptr}) \}$

$F \longrightarrow \texttt{(}\ E\ \texttt{)} \{ \text{F.ptr} := \text{E.ptr} \}$

$F \longrightarrow \texttt{const} \{ \text{F.ptr} := \text{make\_leaf}(\text{const.ptr}) \}$

Figure 4.9: **LL(1) grammar with action routines to build a syntax tree.**

Space Management for Attributes

- **Entries in the attributes stack are pushed and popped automatically**

program ⟶ *stmt_list* $$

stmt_list ⟶ *stmt_list decl* | *stmt_list stmt* | ϵ

decl ⟶ `int id` | `real id`

stmt ⟶ `id :=` *expr* | `read id` | `write` *expr*

expr ⟶ *term* | *expr add_op term*

term ⟶ *factor* | *term mult_op factor*

factor ⟶ `(` *expr* `)` | `id` | `int_const` | `real_const` | `float(` *expr* `)` | `trunc(` *expr* `)`

add_op ⟶ `+` | `-`

mult_op ⟶ `*` | `/`

Figure 4.10: **Context-free grammar for a calculator language with types and declarations.** The intent is that every identifier be declared before use, and that types not be mixed in computations.

Decorating a Syntax Tree

- **Syntax tree for a simple program to print an average of an integer and a real**

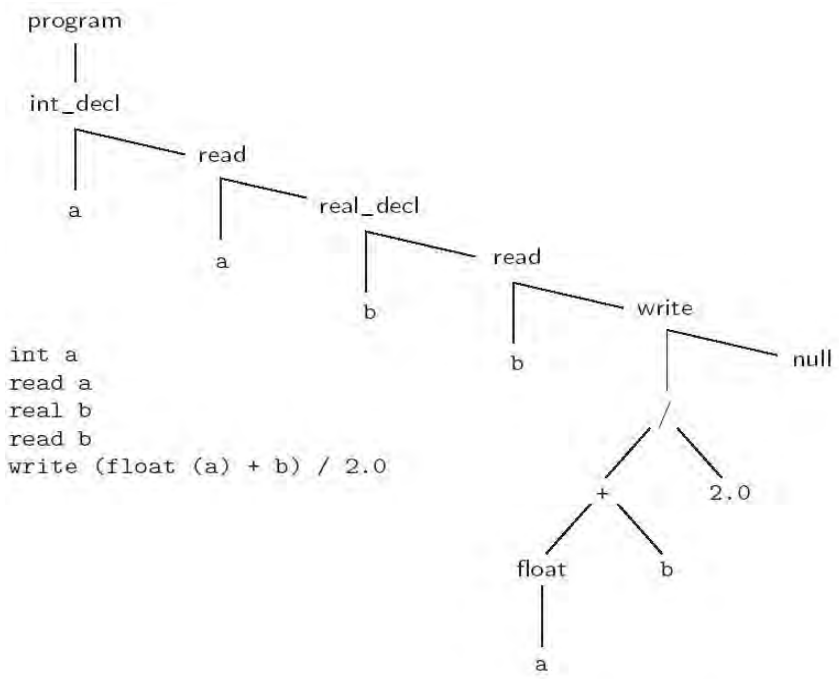


Figure 4.11: **Syntax tree for a simple calculator program.**

- **Tree grammar representing structure of syntax tree**

$$program \longrightarrow item$$
$$int\_decl : item \longrightarrow id\ item$$
$$read : item \longrightarrow id\ item$$
$$real\_decl : item \longrightarrow id\ item$$
$$write : item \longrightarrow expr\ item$$
$$null : item \longrightarrow \epsilon$$
$$\text{`}/\text{'} : expr \longrightarrow expr\ expr$$
$$\text{`}+\text{'} : expr \longrightarrow expr\ expr$$
$$float : expr \longrightarrow expr$$
$$id : expr \longrightarrow \epsilon$$
$$real\_const : expr \longrightarrow \epsilon$$

- **Sample of <u>complete</u> tree grammar representing structure of syntax tree**

$id : expr \longrightarrow \epsilon$
  ▷ if ⟨id.name, A⟩ ∈ expr.symtab -- for some type A
      expr.errors := nil
      expr.type := A
    else
      expr.errors := [id.name "undefined at" id.location]
      expr.type := error

$int\_const : expr \longrightarrow \epsilon$
  ▷ expr.type := int

$real\_const : expr \longrightarrow \epsilon$
  ▷ expr.type := real

$`+' : expr_1 \longrightarrow expr_2\ expr_3$
  ▷ $expr_2$.symtab := $expr_1$.symtab
  ▷ $expr_3$.symtab := $expr_1$.symtab
  ▷ check_types($expr_1$, $expr_2$, $expr_3$)

$`-' : expr_1 \longrightarrow expr_2\ expr_3$
  ▷ $expr_2$.symtab := $expr_1$.symtab
  ▷ $expr_3$.symtab := $expr_1$.symtab
  ▷ check_types($expr_1$, $expr_2$, $expr_3$)

$`\times' : expr_1 \longrightarrow expr_2\ expr_3$
  ▷ $expr_2$.symtab := $expr_1$.symtab
  ▷ $expr_3$.symtab := $expr_1$.symtab
  ▷ check_types($expr_1$, $expr_2$, $expr_3$)

$`\div' : expr_1 \longrightarrow expr_2\ expr_3$
  ▷ $expr_2$.symtab := $expr_1$.symtab
  ▷ $expr_3$.symtab := $expr_1$.symtab
  ▷ check_types($expr_1$, $expr_2$, $expr_3$)

$float : expr_1 \longrightarrow expr_2$
  ▷ $expr_2$.symtab := $expr_1$.symtab
  ▷ convert_type($expr_2$, $expr_1$, int, real, "float of non-int")

$trunc : expr_1 \longrightarrow expr_2$
  ▷ $expr_2$.symtab := $expr_1$.symtab
  ▷ convert_type($expr_2$, $expr_1$, real, int, "trunc of non-real")

UNIT-4

Control Flow

- Basic paradigms for control flow:
  - Sequencing
  - Selection
  - Iteration
  - Procedural Abstraction
  - Recursion
  - Concurrency
  - Exception Handling and Speculation
  - Non-determinacy
- Basic paradigms for control flow:
  - Sequencing: order of execution
  - Selection (also alternation): generally in the form of if or case statements
  - Iteration: loops
  - Recursion: expression is defined in terms of (simpler versions of) itself
  - Nondeterminacy: order or choice is deliberately left unspecified

Expression Evaluation

Precedence, associativity (see Figure 6.1 on next slide)

- C has 15 levels - too many to remember
- Pascal has 3 levels - too few for good semantics
- Fortran has 8
- Ada has 6
  - Ada puts *and & or* at same level
- Lesson: when unsure, use parentheses!

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Infix, Postfix and Prefix

- Prefix: op a b or op(a,b)
  - Example: standard in most we have seen
  - In Lisp: (* (+ 1 3) 2)
- Infix: a op b
  - Sometimes just “syntatic sugar”; for example, in C++, a + b really calls operator+(a,b)
- Postfix: a b op
  - Least common - used in Postscript, Forth, and intermediate code of some compilers
  - Also appears in C (and its descendants) and Pascal examples, such as ++value in C and the pointer dereferencing operator (^) in Pascal

Expression Evaluation

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
  - Commutativity is assumed to be safe
  - associativity (known to be dangerous)

a + (b + c) works if a~=maxint and b~=minint and c<0

(a + b) + c does not

  - This type of operation can be useful, though, for code optimization

- Short-circuiting
  - Consider (a < b) && (b < c):
    - If a >= b there is no point evaluating whether b < c because (a < b) && (b < c) is automatically false
  - Other similar situations

    if (b != 0 && a/b == c) ...

    if (*p && p->foo) ...

  if (unlikely_condition && very_expensive function()) ...

- To be cautious - need to be sure that your second half is valid, or else coder could miss a runtime error without proper testing.
- Variables as values vs. variables as references
  - value-oriented languages
    - C, Pascal, Ada
  - reference-oriented languages
    - most functional languages (Lisp, Scheme, ML)
    - Clu, Smalltalk
  - Algol-68 is halfway in-between
  - Java deliberately in-between
    - built-in types are values
    - user-defined types are objects - references

Expression versus statements

- Most languages distinguish between expressions and statements.
  - Expressions always produce a value, and may or may not have a side effect.
    - Example: In python, b + c
  - Statements are executed solely for their side effects, and return no useful value
    - Example: in Python, mylist.sort()
- A construct has a side effect if it influences subsequent computation in some way (other than simply returning a value).

Expression Evaluation

- Expression-oriented vs. statement-oriented languages
  - expression-oriented:
    - functional languages (Lisp, Scheme, ML)
    - Algol-68
  - statement-oriented:
    - most imperative languages
  - C halfway in-between (distinguishes)
    - allows expression to appear instead of statement, but not the reverse

Algol 68

- Orthogonality
  - Features that can be used in any combination
  - Algol 68 is primary example. Here, everything is an expression (and there is no separate notion of statements).
    - Example:

```
begin
    a := if b<c then d else e;
    a := begin f(b); g(c); end;
    g(d);
    2+3;    end
```

Assignment shortcuts

- Assignment
  - statement (or expression) executed for its side effect - key to most programming languages you have seen so far.
  - assignment operators (+=, -=, etc)
    - Handy shortcuts
    - avoid redundant work (or need for optimization)
    - perform side effects exactly once
      - Example: A[index_fn(i)]++;
      - versus A[index_fn(i)] = A[index_fn(i)] + 1;

C and assignments within expressions

- Combining expressions with assignments can have unfortunate side effects, depending on the language.
  - Pathological example: C has no true boolean type (just uses ints or their equivalents), and allows assignments within expressions.
  - Example:
    - if (a =b) {

…

}

What does this do?

Side effects and functions

- Side Effects
  - often discussed in the context of functions
  - a side effect is some permanent state change caused by execution of function
    - some noticable effect of call other than return value
    - in a more general sense, assignment statements provide the ultimate example of side effects
      - they change the value of a variable

Code optimization

- Most compilers attempt to optimize code:
  - Example: a = b+c, then d = c + e + b
- This can really speed up code:
  - a = b/c/d then e = f/d/c versus
  - t = c*d and then a = b/t and e = f/t
- Arithmetic overflow can really become a problem here.
  - Can be dependent on implementation and local setup
  - Checking provides more work for compiler, so slower
  - With no checks, these can be hard to find

Sequencing

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neuman
- In assembly, the only way to "jump" around is to use branch statements.
- Early programming languages mimicked this, such as Fortran (and even Basic and C).

Sequencing

- Blocks of code are executed in a sequence.
- Block are generally indicated by { … } or similar construct.
- Interesting note: without side effects (as in Agol 68), blocks are essentially useless - the value is just the last return
- In other languages, such as Euclid and Turing, functions which return a value are not allowed to have a side effect at all.
  - Main advantage: these are idempotent - any function call will have the same value, no matter when it occurs
- Clearly, that is not always desirable, of course. (Think of the rand function, which should definitely not return the same thing every time!)

Selection

- Selection
  - Fortran computed gotos
  - jump code
    - for selection and logically-controlled loops
    - no point in computing a Boolean value into a register, then testing it
    - instead of passing register containing Boolean out of expression as a synthesized attribute, pass inherited attributes INTO expression indicating where to jump to if true, and where to jump to if false

Code generated w/ short-circuiting (C)

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1

        L4:     r1 := E

                r2 := F
        if r1 = r2 goto L2
L1:     then_clause
        goto L3
L2:     else_clause
L3:
```

Selection: Case/switch

- The case/switch statement was introduced in Algol W to simplify certain if-else situations.
- Useful when comparing the same integer to a large variety of possibilities:
  - i := (complex expression)

if i == 1: …

elsif i in 2,7: …

  - Case (complex expression)

1: …

2-7: …

Iteration

- Ability to perform some set of operations repeatedly.
  - Loops
  - Recursion
- Can think of iteration as the only way a function won't run in linear time.
- In a real sense, this is the most powerful component of programming.
- In general, loops are more common in imperative languages, while recursion is more common in functional languages.
- Enumeration-controlled: originated in Fortran
  - Pascal or Fortran-style for loops

```
do i = 1, 10, 2
…
enddo
```

  - Changed to standard for loops later, eg Modula-2

```
FOR i := first TO last BY step DO
        …
END
```

Iteration: code generation

- At its heart, none of these initial loops allow anything other than enumeration over a preset, fixed number of values.
- This allows very efficient code generation:

```
    R1 := first
    R2 := step
    R3 := step
L1: …  --loop body, use R1 for I
    R1 := R1 + R2
L2: if R1 <= R2 goto L1
```

Recursion

- Recursion
  - equally powerful to iteration
  - mechanical transformations back and forth
  - often more intuitive (sometimes less)
  - *naïve* implementation less efficient
    - no special syntax required
    - fundamental to functional languages like Scheme
- Tail recursion
  - No computation follows recursive call

```
int gcd (int a, int b) {
        /* assume a, b > 0 */
    if (a == b) return a;
        else if (a > b) return gcd (a - b,b);
        else return gcd (a, b – a);
}
```

  - A good compiler will translate this to machine code that runs “in place”, essentially returning to the start of the function with new a,b values.

Order of evaluation

- Generally, we assume that arguments are evaluated before passing to a subroutine, in applicative order evaluations.
- Not always the case: lazy evaluation or normal order evaluation. pass unevaluated arguments to functions, and value is only computed if and when it is necessary.
- Applicative order is preferable for clarity and efficiency, but sometimes normal order can lead to faster code or code that won’t give as many run-time errors.
- In particular, for list-type structures in functional languages, this lazy evaluation can be key.