PRINCIPLES OF PROGRAMMING LANGUAGES (23IT508)

M.NAGA TRIVENI
Asst. Professor, Department of IT,
NRCM.

NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India
UGC - Autonomous Institute
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH
NRCM
Your roots to success....

UNIT-1

- Preliminaries
- Syntax and Semantics

CONCEPTS

Reasons for Studying Concepts of Programming Languages.

Programming Domains Language Evaluation

Criteria

Influences on Language Design

Language Categories Language Design Trade-

Offs

Implementation Methods Programming

Environments

CONCEPTS

Introduction to syntax and semantics

The General Problem of Describing Syntax

Formal Methods of Describing Syntax

Attribute Grammars

Describing the Meanings of Programs: Dynamic Semantics

❖ *Reasons for Studying Concepts of Programming Languages*

Increased ability to express ideas.

Improved background for choosing appropriate languages.

Increased ability to learn new languages.

Better understanding of significance of implementation.

Better use of languages that are already known. Overall advancement of computing.

❖ Programming Domains

Scientific Applications

– Large numbers of floating point computations; use of arrays.
– Example:Fortran.

Business Applications

– Produce reports, use decimal numbers and characters.
– Example:COBOL.

Artificial intelligence

– Symbols rather than numbers manipulated; use of linked lists.
– Example:LISP.

❖ ***Programming Domains***

System programming

Need effieciency because of continous use.
Example:C

Web Software

-Eclectic collection of languages: Markup(example: XHTML), scripting(example:PHP), general-purpose(example:JAVA).

❖ *Language Evaluation Criteria*

Readability:

The ease with which programs can be read and understood.

Writability:

The ease with which a language can be used to create programs.

Reliability:

Conformance to specifications (i.e., performs to its specifications).

Cost:

- The ultimate total cost.

❖ *Evaluation Criteria: Readability*

Overall simplicity

A manageable set of features and constructs.
Minimal feature multiplicity .
Minimal operator overloading.

Orthogonality

A relatively small set of primitive constructs can be combined in a relatively small number of ways
Every possible combination is legal

Data types

Adequate predefined data types.

❖ Evaluation Criteria:Readability

Syntax considerations

-Identifier forms:flexible composition. - Special words and methods of forming compound statements.

-Form and meaning:self-descriptive constructs,meaningful keywords.

❖ *Evaluation Criteria: Writability*

Simplicity and orthogonality

– Few constructs, a small number of primitives, a small set of rules for combining them.

Support for abstraction

-The ability to define and use complex structures or operations in ways that allow details to be ignored.

Expressivity

– A set of relatively convenient ways of specifying operations.
– Strength and number of operators and predefined functions.

❖ *Evaluation Criteria: Reliability*

Type checking

– Testing for type errors.

Exception handling

– Intercept run-time errors and take corrective measures.

Aliasing

– Presence of two or more distinct referencing methods for the same memory location.

Readability and writability

– A language that does not support "natural" ways of expressing an algorithm will require the use of "unnatural" approaches, and hence reduced reliability.

❖ *Evaluation Criteria: Cost*

Training programmers to use the language

Writing programs (closeness to particular applications)

Compiling programs Executing programs

Language implementation system: availability of free compilers

Reliability: poor reliability leads to high costs

Maintaining programs

Evaluation Criteria: Others

Portability

– The ease with which programs can be moved from one implementation to another.

Generality

– The applicability to a wide range of applications.

Well-definedness

– The completeness and precision of the language’s official definition.

❖ *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

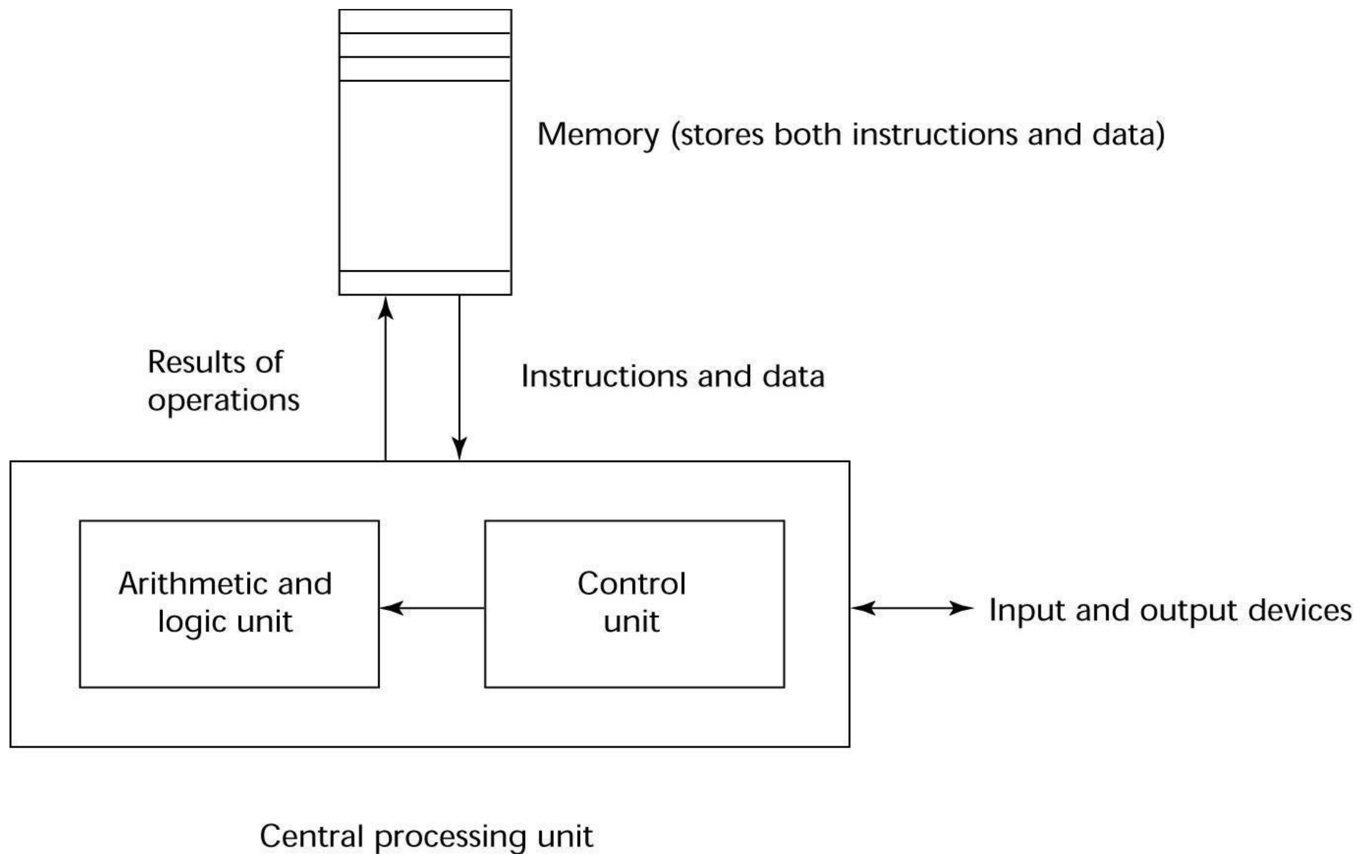
❖ *Computer Architecture Influence*

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



❖ *The Von Neumann Architecture*

Fetch-execute-cycle (on a von Neumann architecture computer)

```
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

❖ *Language Categories*

Imperative

- Central features are variables, assignment statements, and iteration
- Include languages that support object-oriented programming
- Include scripting languages
- Include the visual languages
- Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++

Functional

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

Logic

- Rule-based (rules are specified in no particular order)
- Example: Prolog

Markup/programming hybrid

- Markup languages extended to support some programming
- Examples: JSTL, XSLT

❖ *Language Design Trade-Offs*

Reliability vs. cost of execution

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

Readability vs. writability

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

– Example: C++ pointers are powerful and very flexible but are unreliable

❖ Implementation Methods

Compilation

– Programs are translated into machine language

Pure Interpretation

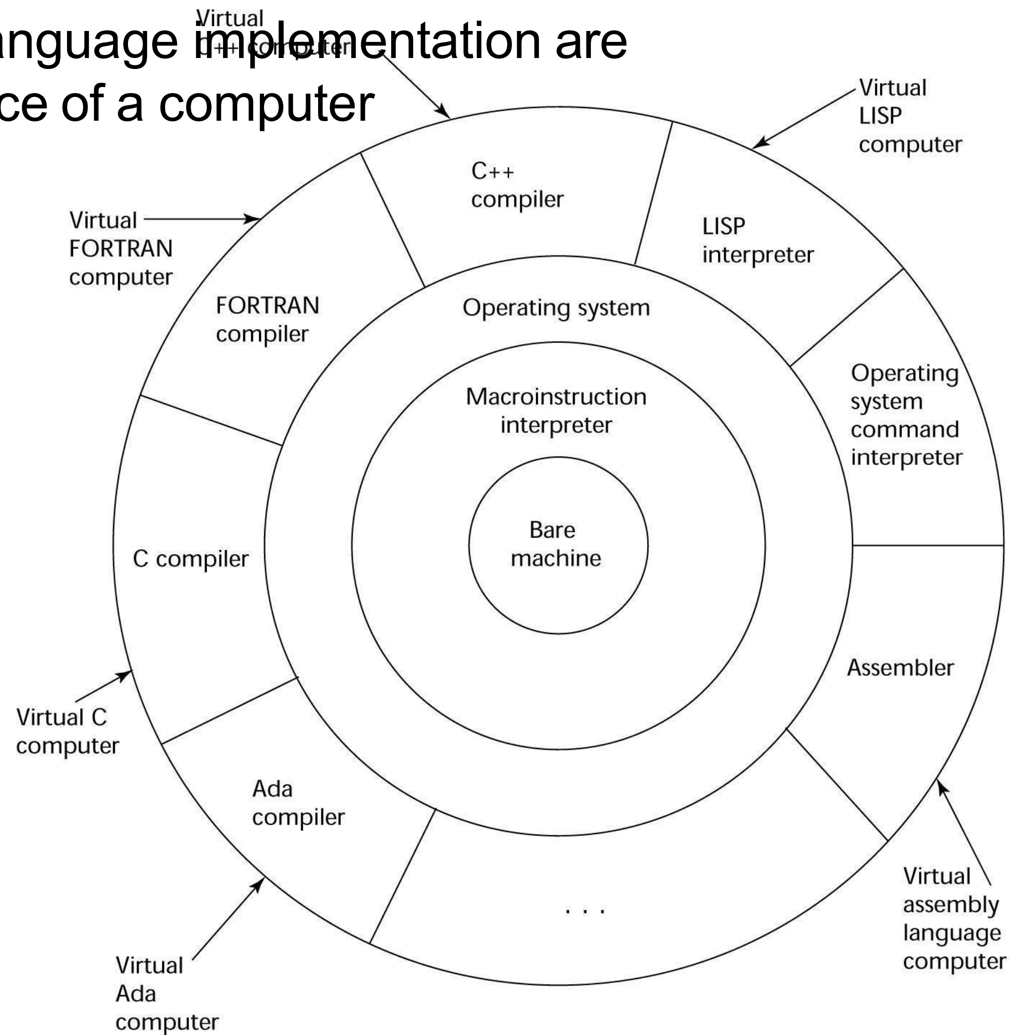
– Programs are interpreted by another program known as an interpreter

Hybrid Implementation Systems

– A compromise between compilers and pure interpreters

❖ *Layered View of Computer*

The operating system and language implementation are layered over machine interface of a computer



Compilation

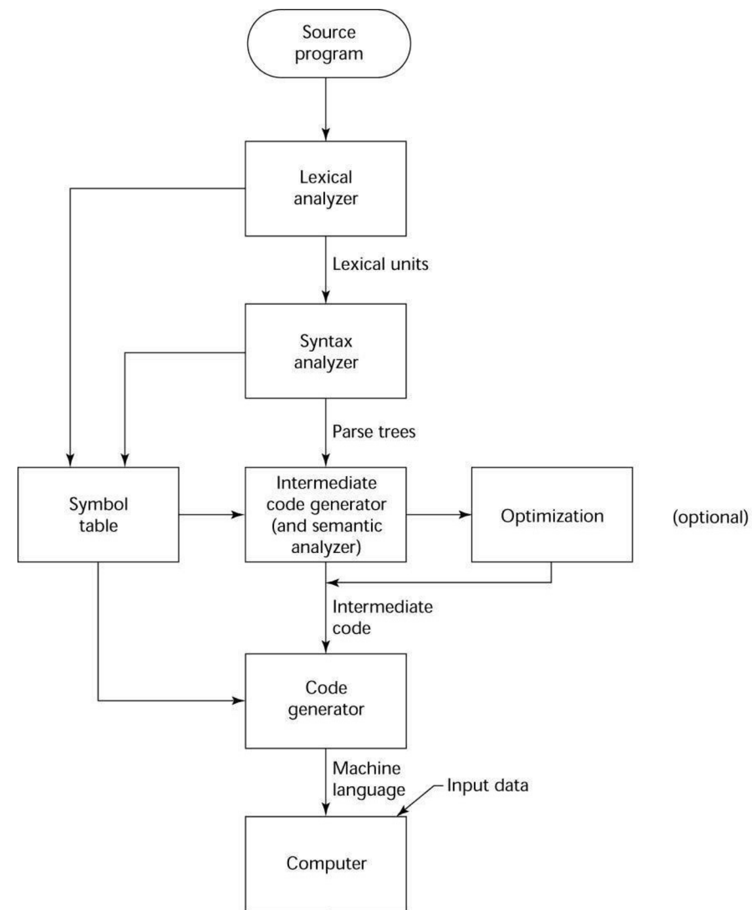
Translate high-level program (source language) into machine code (machine language)

Slow translation, fast execution

Compilation process has several phases:

- lexical analysis: converts characters in the source program into lexical units
- syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
- Semantics analysis: generate intermediate code
- code generation: machine code is generated

The CompilationProcess



Additional Compilation Terminologies

Load module (executable image): the user and system code together

Linking and loading: the process of collecting system program units and linking them to a user program

Von Neumann Bottleneck

Connection speed between a computer's memory and its processor determines the speed of a computer

Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*

Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

Pure Interpretation

No translation

Easier implementation of programs (run-time errors can easily and immediately be displayed)

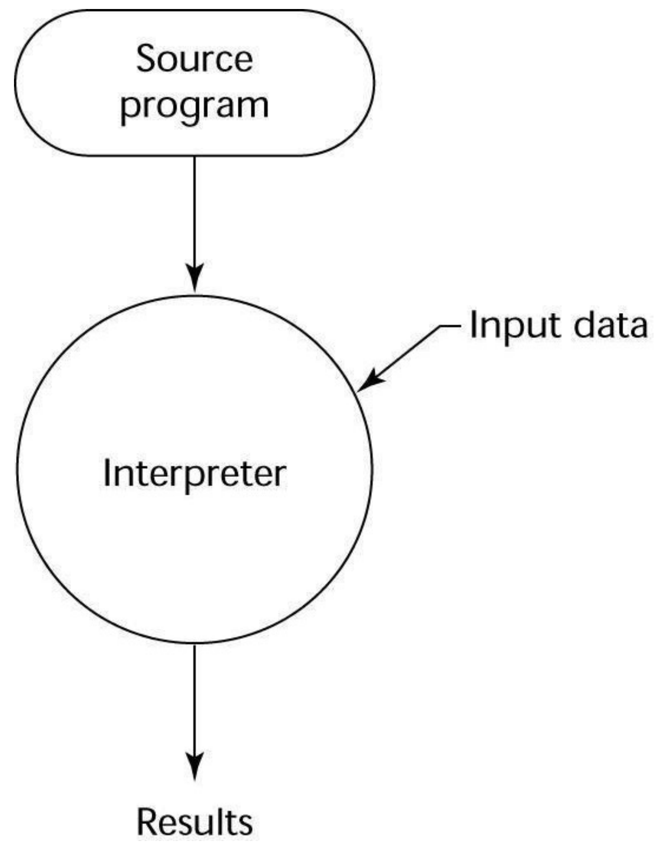
Slower execution (10 to 100 times slower than compiled programs)

Often requires more space

Now rare for traditional high-level languages

Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process



Hybrid Implementation Systems

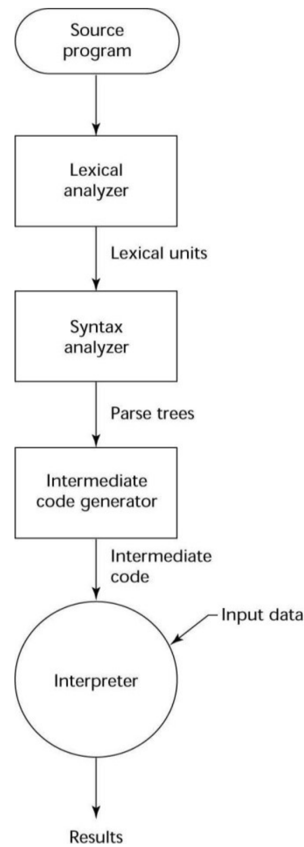
A compromise between compilers and pure interpreters

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

Faster than pure interpretation Examples

- Perl programs are partially compiled to detect errors before interpretation
- Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- 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

Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included

A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros

A well-known example: C preprocessor

– expands `#include`, `#define`, and similar macros

Programming Environments

A collection of tools used in software development

UNIX

- An older operating system and tool collection
- Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX

Microsoft Visual Studio.NET

- A large, complex visual environment

Used to build Web applications and non-Web applications in any .NET language

NetBeans

- Related to Visual Studio .NET, except for Web applications in Java

Zuse's Plankalkül

Minimal Hardware Programming: Pseudocodes

The IBM 704 and Fortran Functional

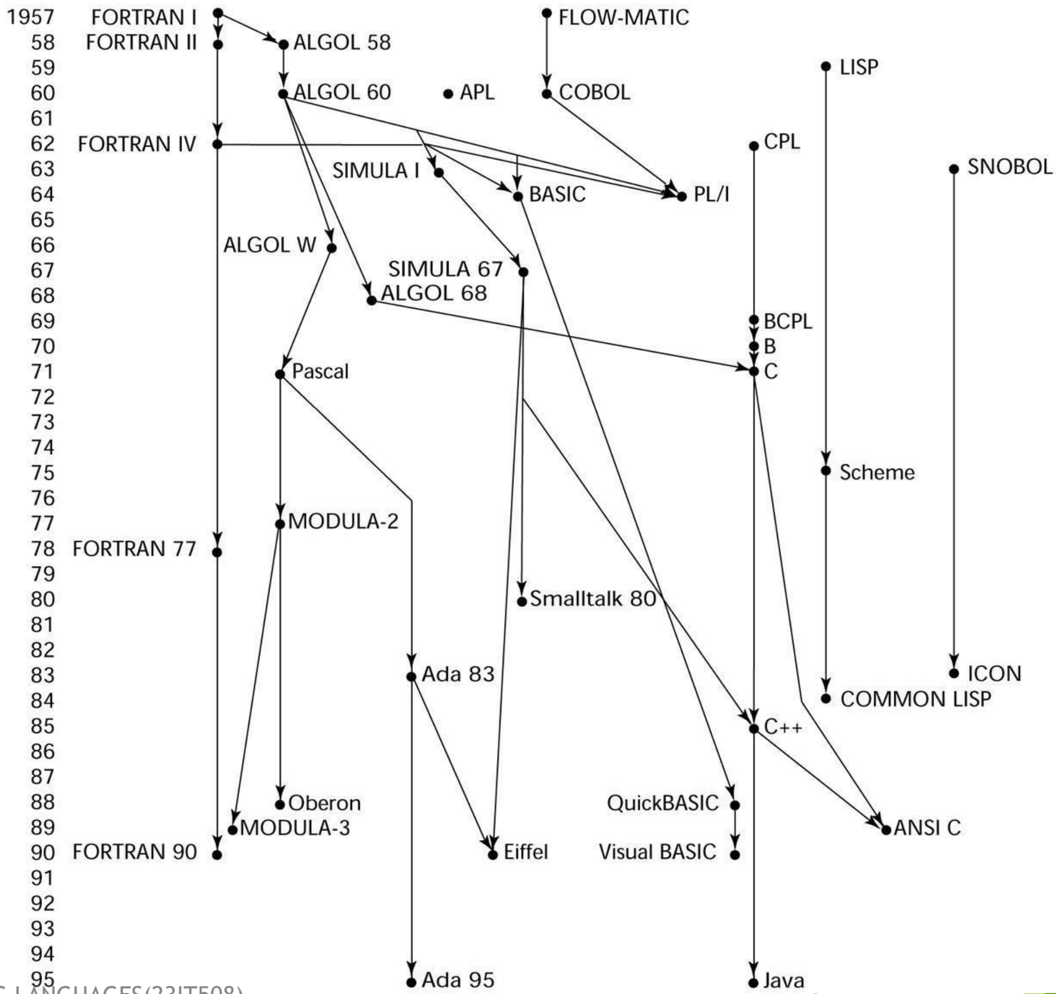
Programming: LISP

The First Step Toward Sophistication: ALGOL 60

Computerizing Business Records: COBOL

The Beginnings of Timesharing: BASIC

Genealogy of Common Languages



Zuse's Plankalkül

- Designed in 1945, but not published until 1972
- Never implemented Advanced data structures
  - – floating point, arrays, records
- Invariants

Plankalkül Syntax

An assignment statement to assign the expression A[4] + 1 to A[5]

```
     |  A + 1 => A
V    |  4        5        (subscripts)
S    |  1.n      1.n      (data types)
```

Minimal Hardware Programming: Pseudocodes

What was wrong with using machine code?

- Poor readability
- Poor modifiability
- Expression coding was tedious
- Machine deficiencies--no indexing or floating point

Pseudocodes: Short Code

Short Code developed by Mauchly in 1949 for BINAC computers

- Expressions were coded, left to right
- Example of operations:

```
01 – 06 abs value 1n (n+2)nd power
0  ) 07  +         2n  (n+2)nd
2                      root
03 = 08 pause       4n if <= n
0  / 09  (         58  print and
4                      tab
```

Pseudocodes: Speedcoding

Speedcoding developed by Backus in 1954 for IBM 701

- Pseudo ops for arithmetic and math functions
- Conditional and unconditional branching
- Auto-increment registers for array access
- Slow!
- Only 700 words left for user program

Pseudocodes: Related Systems

The UNIVAC Compiling System

– Developed by a team led by Grace Hopper

– Pseudocode expanded into machine code

David J. Wheeler (Cambridge University)

– developed a method of using blocks of re-locatable addresses to solve the problem of absolute addressing

IBM 704 and Fortran

Fortran 0: 1954 - not implemented
Fortran I:1957

– Designed for the new IBM 704, which had index registers and floating point hardware

- This led to the idea of compiled programming languages, because there was no place to hide the cost of interpretation (no floating-point software)

– Environment of development Computers were small and unreliable
Applications were scientific
No programming methodology or tools
Machine efficiency was the most important concern

Design Process of Fortran

Impact of environment on design of Fortran I

– No need for dynamic storage
– Need good array handling and counting loops
– No string handling, decimal arithmetic, or powerful input/output (for business software)

Fortran I Overview

First implemented version of Fortran

- Names could have up to six characters
- Post-test counting loop (**DO**)
- Formatted I/O
- User-defined subprograms
- Three-way selection statement (arithmetic **IF**)
- No data typing statements

Fortran I Overview (continued)

First implemented version of FORTRAN

- No separate compilation
- Compiler released in April 1957, after 18 worker-years of effort
- Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of 704
- Code was very fast
- Quickly became widely used

Fortran II

Distributed in 1958

- Independent compilation
- Fixed the bugs

Fortran IV

Evolved during 1960-62

- Explicit type declarations
- Logical selection statement
- Subprogram names could be parameters
- ANSI standard in 1966

Fortran 77

Became the new standard in 1978

- Character string handling
- Logical loop control statement
- **`IF-THEN-ELSE`** statement

Fortran 90

Most significant changes from Fortran 77

- Modules
- Dynamic arrays
- Pointers
- Recursion
- **`CASE`** statement
- Parameter type checking

Latest versions of Fortran

- Fortran 95 – relatively minor additions, plus some deletions
- Fortran 2003 - ditto

Fortran

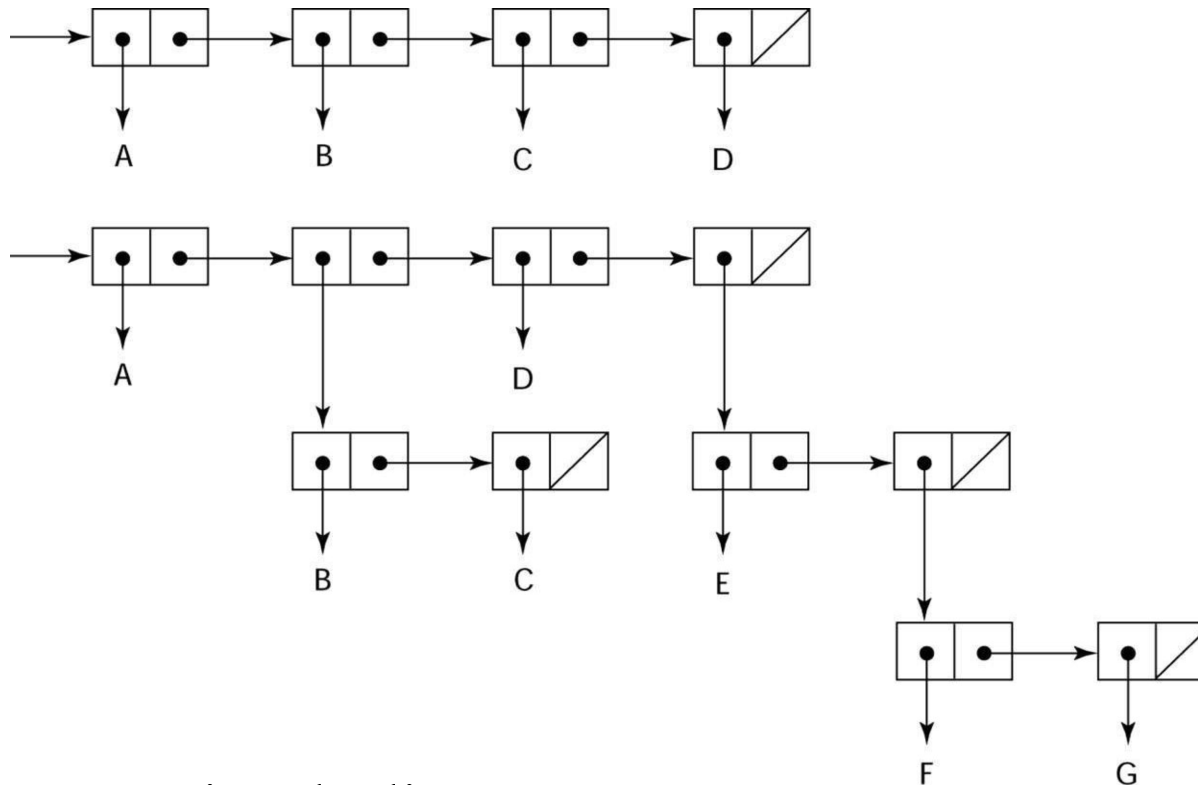
Evaluation

- Highly optimizing compilers (all versions before 90)
  - Types and storage of all variables are fixed before run time
- Dramatically changed forever the way computers are used
- Characterized as the *lingua franca* of the computing world

Functional Programming: LISP

- LISt Processing language
  - Designed at MIT by McCarthy
- AI research needed a language to
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists
- Syntax is based on *lambda calculus*

Representation of Two LISP Lists



Representing the lists `(A B C D)` and `(A (B C) D (E (F G)))`

LISP Evaluation

- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Still the dominant language for AI
- COMMON LISP and Scheme are contemporary dialects of LISP
- ML, Miranda, and Haskell are related languages

Scheme

- Developed at MIT in mid 1970s Small
- Extensive use of static scoping

Functions as first- class entities

- Simple syntax (and small size) make it ideal for educational applications

COMMON LISP

- An effort to combine features of several dialects of LISP into a single language
- Large, complex

The First Step Toward Sophistication: ALGOL 60

Environment of development

- FORTRAN had (barely) arrived for IBM 70x
- Many other languages were being developed, all for specific machines
- No portable language; all were machine-dependent
- No universal language for communicating algorithms

ALGOL 60 was the result of efforts to design a universal language

Early Design Process

ACM and GAMM met for four days for design (May 27 to June 1, 1958)

Goals of the language

- Close to mathematical notation
- Good for describing algorithms
- Must be translatable to machine code

ALGOL 58

Concept of type was formalized Names could be any length

Arrays could have any number of subscripts

Parameters were separated by mode (in & out)

Subscripts were placed in brackets

Compound statements (**begin ... end**)

Semicolon as a statement separator

Assignment operator was :=

if had an **else-if** clause

No I/O - “would make it machine dependent”

ALGOL 58

Implementation

- Not meant to be implemented, but variations of it were (MAD, JOVIAL)
- Although IBM was initially enthusiastic, all support was dropped by mid 1959

ALGOL 60 Overview

Modified ALGOL 58 at 6-day meeting in Paris

New features

- Block structure (local scope)
- Two parameter passing methods
- Subprogram recursion
- Stack-dynamic arrays

- Still no I/O and no string handling

ALGOL 60 Evaluation

Successes

- It was the standard way to publish algorithms for over 20 years
- All subsequent imperative languages are based on it
- First machine-independent language
- First language whose syntax was formally defined (BNF)

ALGOL 60 Evaluation (continued)

Failure

– Never widely used, especially in U.S.

– Reasons

Lack of I/O and the character set made programs non-portable

Too flexible--hard to implement Entrenchment of Fortran

Formal syntax description Lack of support from IBM

Computerizing Business Records: COBOL

Environment of development

- UNIVAC was beginning to use FLOW-MATIC
- USAF was beginning to use AIMACO
- IBM was developing COMTRAN

COBOL Historical Background

Based on FLOW-MATIC FLOW-MATIC features

- Names up to 12 characters, with embedded hyphens
- English names for arithmetic operators (no arithmetic expressions)
- Data and code were completely separate
- The first word in every statement was a verb

COBOL Design Process

First Design Meeting (Pentagon) - May 1959 Design goals

- Must look like simple English
- Must be easy to use, even if that means it will be less powerful
- Must broaden the base of computer users
- Must not be biased by current compiler problems

Design committee members were all from computer manufacturers and DoD branches

Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

COBOL Evaluation

Contributions

– First macro facility in a high-level language
– Hierarchical data structures (records)
– Nested selection statements
– Long names (up to 30 characters), with hyphens
– Separate data division

COBOL: DoD Influence

- First language required by DoD
  - – would have failed without DoD
- Still the most widely used business applications language

The Beginning of Timesharing: BASIC

Designed by Kemeny & Kurtz at Dartmouth

Design Goals:

- Easy to learn and use for non-science students
- Must be "pleasant and friendly"
- Fast turnaround for homework
- Free and private access
- User time is more important than computer time

Current popular dialect: Visual BASIC

First widely used language with time sharing

2.8 Everything for Everybody: PL/I

Designed by IBM and SHARE

Computing situation in 1964 (IBM's point of view)

- Scientific computing
  - IBM 1620 and 7090 computers FORTRAN
  - SHARE user group
- Business computing
  - IBM 1401, 7080 computers COBOL
  - GUIDE user group

PL/I: Background

By 1963

- Scientific users began to need more elaborate I/O, like COBOL had; business users began to need floating point and arrays for MIS
- It looked like many shops would begin to need two kinds of computers, languages, and support staff--too costly

The obvious solution

- Build a new computer to do both kinds of applications
- Design a new language to do both kinds of applications

PL/I: Design Process

- Designed in five months by the 3 X 3 Committee
  - Three members from IBM, three members from SHARE
- Initial concept
  - An extension of Fortran IV
- Initially called NPL (New Programming Language)
- Name changed to PL/I in 1965

PL/I: Evaluation

PL/I contributions

- First unit-level concurrency
- First exception handling
- Switch-selectable recursion
- First pointer data type
- First array cross sections

Concerns

- Many new features were poorly designed
- Too large and too complex

Two Early Dynamic Languages: APL and SNOBOL

- Characterized by dynamic typing and dynamic storage allocation
- Variables are untyped
  - A variable acquires a type when it is assigned a value
- Storage is allocated to a variable when it is assigned a value

APL: A Programming Language

Designed as a hardware description language at IBM by Ken Iverson around 1960

- Highly expressive (many operators, for both scalars and arrays of various dimensions)
- Programs are very difficult to read

Still in use; minimal changes

SNOBOL

- Designed as a string manipulation language at Bell Labs by Farber, Griswold, and Polensky in 1964
- Powerful operators for string pattern matching
- Slower than alternative languages (and thus no longer used for writing editors)
- Still used for certain text processing tasks

The Beginning of Data Abstraction: SIMULA 67

Designed primarily for system simulation in Norway by Nygaard and Dahl

Based on ALGOL 60 and SIMULA I

Primary Contributions

- Coroutines - a kind of subprogram
- Classes, objects, and inheritance

Orthogonal Design: ALGOL 68

- From the continued development of ALGOL 60 but not a superset of that language
- Source of several new ideas (even though the language itself never achieved widespread use)
- Design is based on the concept of orthogonality
  - – A few basic concepts, plus a few combining mechanisms

ALGOL 68 Evaluation

Contributions

– User-defined data structures

– Reference types

– Dynamic arrays (called flex arrays)

Comments

– Less usage than ALGOL 60

– Had strong influence on subsequent languages, especially Pascal, C, and Ada

Pascal - 1971

- Developed by Wirth (a former member of the ALGOL 68 committee)
- Designed for teaching structured programming
- Small, simple, nothing really new
- Largest impact was on teaching programming
  – From mid-1970s until the late 1990s, it was the most widely used language for teaching programming

C - 1972

- Designed for systems programming (at Bell Labs by Dennis Richie)
- Evolved primarily from BCLP, B, but also ALGOL 68
- Powerful set of operators, but poor type checking
- Initially spread through UNIX Many areas of application

Programming Based on Logic: Prolog

Developed, by Comerauer and Roussel (University of Aix-Marseille), with help from Kowalski (University of Edinburgh)

Based on formal logic Non-procedural

Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries Highly inefficient, small application areas

History's Largest Design Effort: Ada

- Huge design effort, involving hundreds of people, much money, and about eight years
  - Strawman requirements (April 1975)
  - Woodman requirements (August 1975)
  - Tinman requirements (1976)
  - Ironman equipments (1977)
  - Steelman requirements (1978)
- Named Ada after Augusta Ada Byron, the first programmer

Ada Evaluation

Contributions

- Packages - support for data abstraction
- Exception handling - elaborate
- Generic program units
- Concurrency - through the tasking model

Comments

- Competitive design
- Included all that was then known about software engineering and language design
- First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed

Ada 95

Ada 95 (began in 1988)

- Support for OOP through type derivation
- Better control mechanisms for shared data
- New concurrency features
- More flexible libraries

Popularity suffered because the DoD no longer requires its use but also because of popularity of C++

Object-Oriented Programming: Smalltalk

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- First full implementation of an object- oriented language (data abstraction, inheritance, and dynamic binding)
- Pioneered the graphical user interface design Promoted OOP

Combining Imperative and Object- Oriented Programming: C++

Developed at Bell Labs by Stroustrup in 1980 Evolved from C and SIMULA 67

Facilities for object-oriented programming, taken partially from SIMULA 67

Provides exception handling

A large and complex language, in part because it supports both procedural and OO programming

Rapidly grew in popularity, along with OOP

ANSI standard approved in November 1997

Microsoft's version (released with .NET in 2002): Managed C++

- delegates, interfaces, no multiple inheritance

Related OOP Languages

- Eiffel (designed by Bertrand Meyer - 1992)
  - Not directly derived from any other language
  - Smaller and simpler than C++, but still has most of the power
  - Lacked popularity of C++ because many C++ enthusiasts were already C programmers
- Delphi (Borland)
  - Pascal plus features to support OOP
  - More elegant and safer than C++

An Imperative-Based Object-Oriented Language: Java

- Developed at Sun in the early 1990s
  - C and C++ were not satisfactory for embedded electronic devices
- Based on C++
  - Significantly simplified (does not include **`struct, union, enum`**, pointer arithmetic, and half of the assignment coercions of C++)
  - Supports *only* OOP
  - Has references, but not pointers
  - Includes support for applets and a form of concurrency

Java Evaluation

Eliminated many unsafe features of C++

Supports concurrency

Libraries for applets, GUIs, database access

Portable: Java Virtual Machine concept, JIT compilers

Widely used for Web programming

Use increased faster than any previous language

Most recent version, 5.0, released in 2004

Scripting Languages for the Web

Perl

- Designed by Larry Wall—first released in 1987
- Variables are statically typed but implicitly declared
- Three distinctive namespaces, denoted by the first character of a variable's name
- Powerful, but somewhat dangerous
- Gained widespread use for CGI programming on the Web
- Also used for a replacement for UNIX system administration language

JavaScript

- Began at Netscape, but later became a joint venture of Netscape and Sun Microsystems
- A client-side HTML-embedded scripting language, often used to create dynamic HTML documents
- Purely interpreted
- Related to Java only through similar syntax

PHP

- PHP: Hypertext Preprocessor, designed by Rasmus Lerdorf
- A server-side HTML-embedded scripting language, often used for form processing and database access through the Web
- Purely interpreted

Scripting Languages for the Web

Python

- An OO interpreted scripting language
- Type checked but dynamically typed
- Used for CGI programming and form processing
- Dynamically typed, but type checked

– Supports lists, tuples, and hashes Lua

- An OO interpreted scripting language
- Type checked but dynamically typed
- Used for CGI programming and form processing
- Dynamically typed, but type checked
- Supports lists, tuples, and hashes, all with its single data structure, the table
- Easily extendable

Scripting Languages for the Web

- Ruby
  - Designed in Japan by Yukihiro Matsumoto (a.k.a, “Matz”)
  - Began as a replacement for Perl and Python
  - A pure object-oriented scripting language
    - All data are objects
  - Most operators are implemented as methods, which can be redefined by user code
  - Purely interpreted

C-Based Language for the New Millennium: C#

Part of the .NET development platform (2000)

Based on C++ , Java, and Delphi

Provides a language for component-based software development

All .NET languages use Common Type System (CTS), which provides a common class library

Markup/Programming Hybrid Languages

XSLT

- eXtensible Markup Language (XML): a metamarkup language
- eXtensible Stylesheet Language Transformation (XSTL) transforms XML documents for display
- Programming constructs (e.g., looping)

JSP

- Java Server Pages: a collection of technologies to support dynamic Web documents
- servlet: a Java program that resides on a Web server and is enacted when called by a requested HTML document; a servlet's output is displayed by the browser
- JSTL includes programming constructs in the form of HTML elements

Introduction to 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
  - Syntax and semantics provide a language's definition
- – Users of a language definition
  - Other language designers Implementers
  - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

A *sentence* is a string of characters over some alphabet

A *language* is a set of sentences

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)

Formal Definition of Languages

Recognizers

– A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
– Example: syntax analysis part of a compiler

Detailed discussion of syntax analysis appears in Chapter 4

Generators

– A device that generates sentences of a language
– One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

BNF and Context-Free Grammars

Context-Free Grammars

- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context- free languages

Backus-Naur Form (1959)

- Invented by John Backus to describe Algol 58
- BNF is equivalent to context-free grammars

BNF Fundamentals

In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *terminals*)

Terminals are lexemes or tokens

A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

Nonterminals are often enclosed in angle brackets

- Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

Grammar: a finite non-empty set of rules

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS
  - <stmt> → <single_stmt>
    - | begin <stmt_list> end

Describing Lists

Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

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

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

Derivations

Every string of symbols in a derivation is a *sentential form*

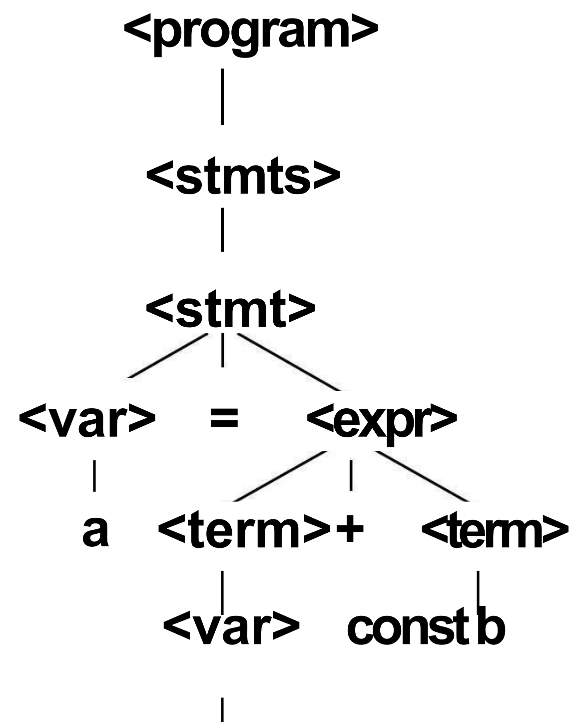
A *sentence* is a sentential form that has only terminal symbols

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

Parse Tree

A hierarchical representation of a derivation

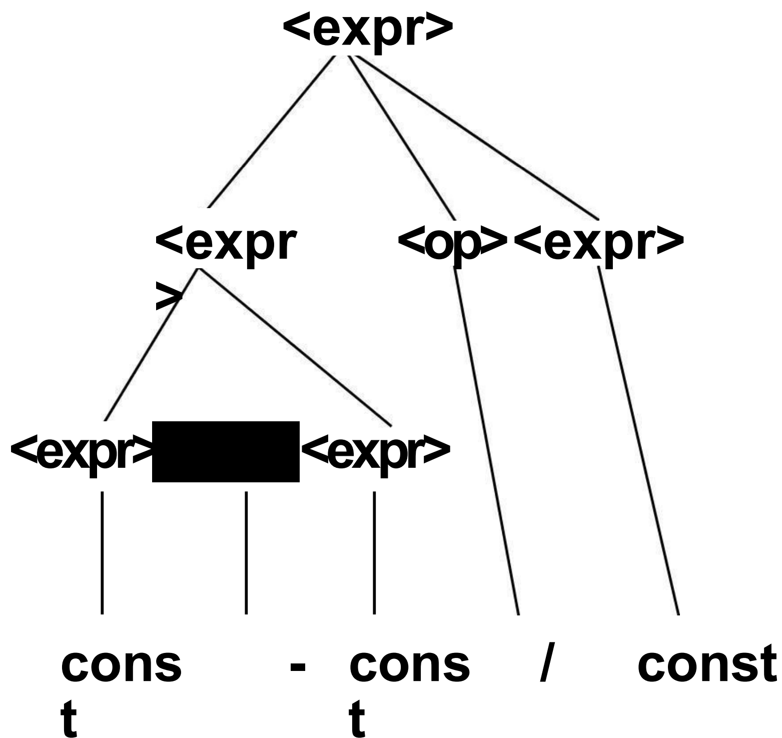


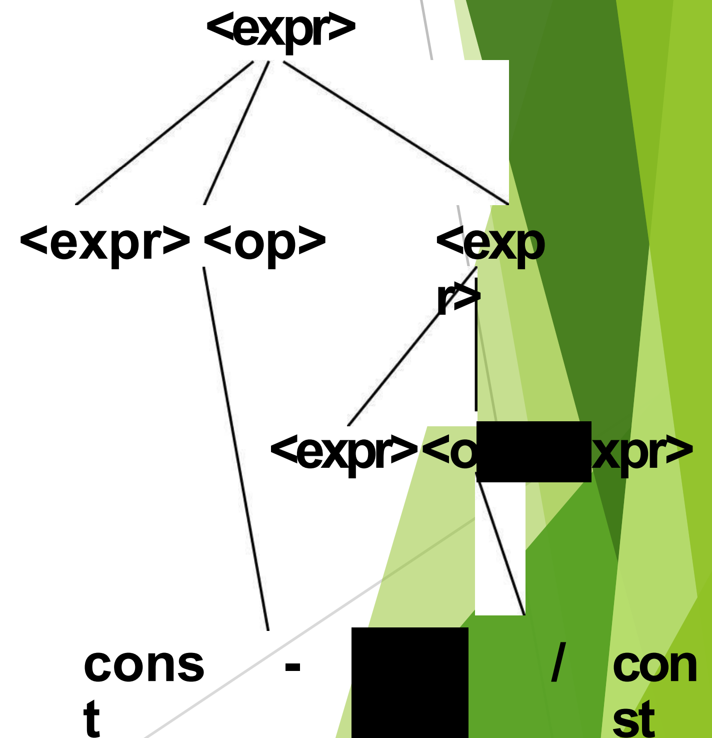
Ambiguity in Grammars

A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

```
<expr> → <expr> <op> <expr> | const
<op> → / | -
```

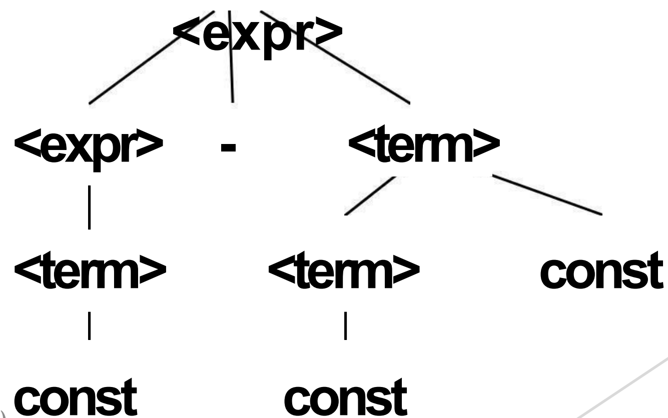




An Unambiguous Expression Grammar

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

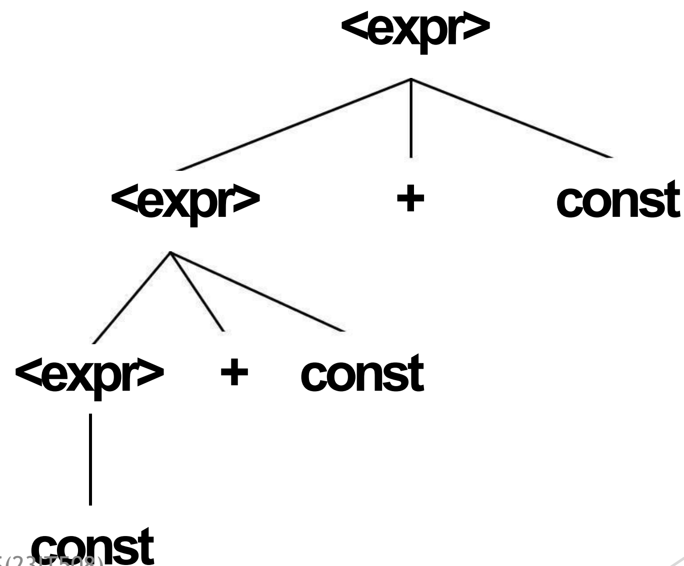
```
<expr> → <expr> - <term> | <term>
<term> → <term> / const| const
```



Associativity of Operators

Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const   (ambiguous)
<expr> -> <expr> + const  |  const   (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]
  `<proc_call> -> ident [(<expr_list>)]`
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  `<term> → <term> (+|-) const`
- Repetitions (0 or more) are placed inside braces { }
  `<ident> → letter {letter|digit}`

BNF and EBNF

BNF

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

EBNF

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

Recent Variations in EBNF

Alternative RHSs are put on separate lines
Use of a colon instead of `=>`
Use of `opt` for optional parts Use of `oneof`
for choices

Static Semantics

- Nothing to do with meaning
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Attribute Grammars

Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

Primary value of AGs:

- Static semantics specification
- Compiler design (static semantics checking)

Attribute Grammars : Definition

Def: An attribute grammar is a context-free grammar G = (S, N, T, P) with the following additions:

- For each grammar symbol *x* there is a set *A(x)* of attribute values
- Each rule has a set of functions that define certain attributes of the nonterminals in the rule
- Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars: Definition

Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule

Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes*

Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $i <= j <= n$, define *inherited attributes*

Initially, there are *intrinsic attributes* on the leaves

Attribute Grammars: An Example

Syntax

<assign> -> <var> = <expr>

<expr> -> <var> + <var> | <var>

<var> -> A | B | C

actual_type: synthesized for <var> and <expr>

expected_type: inherited for <expr>

Attribute Grammar (continued)

Syntax rule: `<expr> → <var>[1] + <var>[2]`

Semantic rules:

`<expr>.actual_type ← <var>[1].actual_type`

Predicate:

```
<var>[1].actual_type == <var>[2].actual_type
<expr>.expected_type == <expr>.actual_type
```

Syntax rule: `<var> → id`

Semantic rule:

`<var>.actual_type ← lookup (<var>.string)`

Attribute Grammars (continued)

How are attribute values computed?

- If all attributes were inherited, the tree could be decorated in top-down order.
- If all attributes were synthesized, the tree could be decorated in bottom-up order.
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Attribute Grammars (continued)

```
<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type
```

Semantics

There is no single widely acceptable notation or formalism for describing semantics

Several needs for a methodology and notation for semantics:

- Programmers need to know what statements mean
- Compiler writers must know exactly what language constructs do
- Correctness proofs would be possible
- Compiler generators would be possible
- Designers could detect ambiguities and inconsistencies

Operational Semantics

Operational Semantics

– Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

To use operational semantics for a high-level language, a virtual machine is needed

Operational Semantics

A *hardware* pure interpreter would be too expensive

A *software* pure interpreter also has problems

- The detailed characteristics of the particular computer would make actions difficult to understand
- Such a semantic definition would be machine-dependent

Operational Semantics (continued)

A better alternative: A complete computer simulation

The process:

– Build a translator (translates source code to the machine code of an idealized computer)

– Build a simulator for the idealized computer

Evaluation of operational semantics:

– Good if used informally (language manuals, etc.)

– Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Operational Semantics (continued)

Uses of operational semantics: Language manuals and textbooks Teaching programming languages

Two different levels of uses of operational semantics:
Natural operational semantics Structural operational semantics

Evaluation
Good if used informally (language manuals, etc.)
- Extremely complex if used formally (e.g.,VDL)

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

Denotational Semantics - continued

The process of building a denotational specification for a language:

Define a mathematical object for each language entity

– Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

The meaning of language constructs are defined by only the values of the program's variables

Denotational Semantics:

program state

The state of a program is the values of all its current variables

$$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

$$VARMAP(i_j, s) = v_j$$

Decimal Numbers

```
<dec_num> → '0' | '1' | '2' | '3' | '4' | '5' |
            '6' | '7' | '8' | '9' |
            <dec_num> ('0' | '1' | '2' | '3'
            | '4' | '5' | '6' | '7'
            | '8' | '9')

Mdec('0') = 0, Mdec ('1') = 1, …, Mdec ('9') = 9

Mdec (<dec_num> '0') = 10 * Mdec (<dec_num>)
Mdec (<dec_num> '1') = 10 * Mdec (<dec_num>) + 1
…
Mdec (<dec_num> '9') = 10 * Mdec (<dec_num>) + 9
```

Expressions

Map expressions onto Z U {error}

We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

Expressio ns

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
            if VARMAP(<var>, s) == undef then error
                  else VARMAP(<var>, s)
     <binary_expr> =>
            if (Me(<binary_expr>.<left_expr>, s) == undef OR
                  Me(<binary_expr>.<right_expr>, s) =
                                   undef)
                  then error
            else
            if (<binary_expr>.<operator> == '+' then
               Me(<binary_expr>.<left_expr>, s) +
                       Me(<binary_expr>.<right_expr>, s)
            else Me(<binary_expr>.<left_expr>, s) *
                Me(<binary_expr>.<right_expr>, s)
        ...
```

Assignment Statements

Maps state sets to state sets U {`error`}

```
Ma(x := E, s) Δ=
    if Me(E, s) == error then error
      else s' =
     {<i1,v1'>,<i2,v2'>,...,<in,vn'>},
            where for j = 1, 2, ..., n,
                if ij  == x
                  then vj' = Me(E, s)
                  else vj' = VARMAP(ij, s)
```

Logical Pretest Loops

Maps state sets to state sets U {error}

```
Ml(while B do L, s) Δ=
    if Mb(B, s) == undef then error
        else if Mb(B, s) == false then s
            else if Msl(L, s) == error then error
                else Ml(while B do L, Msl(L, s))
```

Loop Meaning

The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors

In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

Recursion, when compared to iteration, is easier to describe with mathematical rigor

Evaluation of Denotational Semantics

Can be used to prove the correctness of programs

Provides a rigorous way to think about programs

Can be an aid to language design

Has been used in compiler generation systems

Because of its complexity, it are of little use to language users

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called *assertions*

Axiomatic Semantics (continued)

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a *postcondition*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

Axiomatic Semantics Form

Pre-, post form: `{P} statement {Q}`

An example

- `a = b + 1  {a > 1}`
- One possible precondition: `{b > 10}`
- Weakest precondition: `{b > 0}`

Program Proof Process

The postcondition for the entire program is the desired result

– Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

Axiomatic Semantics: Axioms

An axiom for assignment statements (x = E): $\{Q_{x->E}\}$ x = E {Q}

The Rule of Consequence:

$$\frac{\{P\}\,S\,\{Q\},\, P' \Rightarrow P,\, Q \Rightarrow Q'}{\{P'\}\,S\,\{Q'\}}$$

Axiomatic Semantics: Axioms

An inference rule for sequences of the form S1; S2

{P1} S1 {P2}
{P2} S2 {P3}

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Axiomatic Semantics: Axioms

An inference rule for logical pretest loops

{P} while B do S end {Q}

$$\frac{\text{(I and B) S \{I\}}}{\text{\{I\} while B do S \{I and (not B)\}}}$$

where I is the loop invariant (the inductive hypothesis)

Axiomatic Semantics: Axioms

Characteristics of the loop invariant: I must meet the following conditions:

- P => I   the loop invariant must be true initially
- {I} B {I}   evaluation of the Boolean must not change the validity of I
- {I and B} S {I}   -- I is not changed by executing the body of the loop
- (I and (not B)) => Q   if I is true and B is false, Q is implied
- The loop terminates   can be difficult to prove

Loop Invariant

The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.

I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotation Semantics vs Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational

- Development, development environment, and evaluation of a number of programming languages

Perspective into current issues in language design

The study of programming languages is valuable for a number of reasons:

- Increase our capacity to use different constructs
- Enable us to choose languages more intelligently
- Makes learning new languages easier

Most important criteria for evaluating programming languages include:

- Readability, writability, reliability, cost

Major influences on language design have been machine architecture and software development methodologies

The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation