

Evolution of programming (cont.)

mid 1950's: assembly languages developed

- mnemonic names replaced numeric codes
- relative addressing via names and labels

a separate program (*assembler*) translated from assembly code to machine code

- still machine specific, low-level

```
.file "hello.cpp"
gcc2_compiled.:
.global _Q_qtod
.section ".rodata"
.align 8
.LLC0: .asciz "Hello world!"
.section ".text"
.align 4
.global main
.type main,#function
.proc 04
main: !#PROLOGUE# 0
save %sp,-112,%sp
!#PROLOGUE# 1
sethi %hi(cout),%o1
or %o1,%lo(cout),%o0
sethi %hi(.LLC0),%o2
or %o2,%lo(.LLC0),%o1
call __ls__7ostreamPcc,0
nop
mov %o0,%l0
mov %l0,%o0
sethi %hi(endl__FR7ostream),%o2
or %o2,%lo(endl__FR7ostream),%o1
call __ls__7ostreamPFR7ostream_R7ostream,0
nop
mov 0,%i0
b .LL230
nop
.LL230: ret
restore
.LLfel: .size main,.LLfel-main
.ident "GCC: (GNU) 2.7.2"
```

3

Evolution of programming (cont.)

late 1950's: high-level languages developed

- allowed user to program at higher level of abstraction

however, bridging the gap to low-level hardware was more difficult

- a *compiler* translated code all at once into machine code (e.g., FORTRAN, C++)
- an *interpreter* simulated execution of the code line-by-line (e.g., BASIC, Scheme)

```
// File: hello.cpp
// Author: Dave Reed
//
// This program prints "Hello world!"
////////////////////////////////////

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello world!" << endl;

    return 0;
}
```

4

Software development methodologies

by 70's, software costs rivaled hardware

→ new development methodologies emerged

early 70's: top-down design

- stepwise (iterative) refinement (Pascal)

late 70's: data-oriented programming

- concentrated on the use of ADT's (Modula-2, Ada, C/C++)

early 80's: object-oriented programming

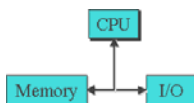
- ADT's+inheritance+dynamic binding (Smalltalk, C++, Eiffel, Java)

mid 90's: extreme programming, agile programming (???)

5

Architecture influences design

virtually all computers follow the *von Neumann* architecture



fetch-execute cycle: repeatedly

- fetch instructions/data from memory
- execute in CPU
- write results back to memory

imperative languages parallel this behavior

- variables (memory cells)
- assignments (changes to memory)
- sequential execution & iteration (fetch/execute cycle)

since features resemble the underlying implementation, tend to be efficient

declarative languages emphasize problem-solving approaches far-removed from the underlying hardware

e.g., Prolog (logic): specify facts & rules, interpreter performs logical inference

LISP/Scheme (functional): specify dynamic transformations to symbols & lists

tend to be more flexible and expressive, but not as efficient

6

FORTRAN (Formula Translator)

FORTRAN was the first* high-level language

- developed by John Backus at IBM
- designed for the IBM 704 computer, all control structures corresponded to 704 machine instructions
- 704 compiler completed in 1957

- despite some early problems, FORTRAN was immensely popular – adopted universally in 50's & 60's

- FORTRAN evolved based on experience and new programming features
 - FORTRAN II (1958)
 - FORTRAN IV (1962)
 - FORTRAN 77 (1977)
 - FORTRAN 90 (1990)

```
C
C  FORTRAN program
C  Prints "Hello world" 10 times
C
      PROGRAM HELLO
      DO 10, I=1,10
         PRINT *, 'Hello world'
      10 CONTINUE
      STOP
      END
```

7

LISP (List Processing)

LISP is a functional language

- developed by John McCarthy at MIT
- designed for Artificial Intelligence research – needed to be symbolic, flexible, dynamic
- LISP interpreter completed in 1959

- LISP syntax is very simple but flexible, based on the λ -calculus of Church
- all memory management is dynamic and automatic – simple but inefficient

- LISP is still the dominant language in AI

- dialects of LISP have evolved
 - Scheme (1975)
 - Common LISP (1984)

```
;;; LISP program
;;; (hello N) will return a list containing
;;;   N copies of "Hello world"

(define (hello N)
  (if (zero? N)
      '()
      (cons "Hello world" (hello (- N 1)))))
```

```
> (hello 10)
("Hello world"
 "Hello world"
 "Hello world"
 "Hello world"
 "Hello world"
 "Hello world"
 "Hello world"
 "Hello world"
 "Hello world"
 "Hello world")
>
```

8

ALGOL (Algorithmic Language)

ALGOL was an international effort to design a universal language

- developed by joint committee of ACM and GAMM (German equivalent)
- influenced by FORTRAN, but more flexible & powerful, not machine specific
- ALGOL introduced and formalized many common language features of today
 - data type
 - compound statements
 - natural control structures
 - parameter passing modes
 - recursive routines
 - BNF for syntax (Backus & Naur)
- ALGOL evolved (58, 60, 68), but not widely adopted as a programming language
 - instead, accepted as a reference language

```
comment ALGOL 60 PROGRAM
  displays "Hello world" 10 times;
begin
  integer counter;
  for counter := 1 step 1 until 10 do
  begin
    printstring(Hello world");
  end
end
```

9

C → C++ → Java → JavaScript

ALGOL influenced the development of virtually all modern languages

- C (1971, Dennis Ritchie at Bell Labs)
 - designed for system programming (used to implement UNIX)
 - provided high-level constructs and low-level machine access
- C++ (1985, Bjarne Stroustrup at Bell Labs)
 - extended C to include objects
 - allowed for object-oriented programming, with most of the efficiency of C
- Java (1993, Sun Microsystems)
 - based on C++, but simpler & more reliable
 - purely object-oriented, with better support for abstraction and networking
- JavaScript (1995, Netscape)
 - Web scripting language

```
#include <stdio.h>
main() {
  for(int i = 0; i < 10; i++) {
    printf ("Hello World!\n");
  }
}
```

```
#include <iostream>
using namespace std;
int main() {
  for(int i = 0; i < 10; i++) {
    cout << "Hello World!" << endl;
  }
  return 0;
}
```

```
class HelloWorld {
  public static void main (String args[]) {
    for(int i = 0; i < 10; i++) {
      System.out.print("Hello World ");
    }
  }
}
```

```
<html>
<body>
  <script language="JavaScript">
    for(i = 0; i < 10; i++) {
      document.write("Hello World<br>");
    }
  </script>
</body>
</html>
```

10

Other influential languages

COBOL (1960, Dept of Defense/Grace Hopper)

- designed for business applications, features for structuring data & managing files

BASIC (1964, Kemeny & Kurtz – Dartmouth)

- designed for beginners, unstructured but popular on microcomputers in 70's

Simula 67 (1967, Nygaard & Dahl – Norwegian Computing Center)

- designed for simulations, extended ALGOL to support classes/objects

Pascal (1971, Wirth – Stanford)

- designed as a teaching language but used extensively, emphasized structured programming

Prolog (1972, Colmerauer, Roussel – Aix-Marseille, Kowalski – Edinburgh)

- logic programming language, programs stated as collection of facts & rules

Ada (1983, Dept of Defense)

- large & complex (but powerful) language, designed to be official govt. contract language 11

There is no silver bullet

remember: there is no *best* programming language

- each language has its own strengths and weaknesses

languages can only be judged within a particular domain or for a specific application

business applications → COBOL

artificial intelligence → LISP/Scheme or Prolog

systems programming → C

software engineering → C++ or Java or Smalltalk

Web development → Java or JavaScript or VBScript or perl

Syntax

syntax: the form of expressions, statements, and program units in a programming language

programmers & implementers need a clear, unambiguous description

formal methods for describing syntax:

- Backus-Naur Form (BNF)
developed to describe ALGOL (originally by Backus, updated by Naur)
allowed for clear, concise ALGOL 60 report
(paralleled grammar work by Chomsky: BNF = context-free grammar)
- Extended BNF (EBNF)
- syntax graphs

13

BNF is a meta-language

a grammar is a collection of rules that define a language

- BNF rules define abstractions in terms of terminal symbols and abstractions

```
<ASSIGN> → <VAR> := <EXPRESSION>
```

- rules can be conditional using '|' to represent OR

```
<IF-STMT> → if <LOGIC-EXPR> then <STMT>  
           | if <LOGIC-EXPR> then <STMT> else <STMT>
```

- arbitrarily long expressions can be defined using recursion

```
<IDENT-LIST> → <IDENTIFIER>  
             | <IDENTIFIER> , <IDENT-LIST>
```

14

Deriving expressions from a grammar

from ALGOL 60:

```
<letter>   → a | b | c | ... | z | A | B | ... | Z
<digit>    → 0 | 1 | 2 | ... | 9
<identifier> → <letter>
              | <identifier> <letter>
              | <identifier> <digit>
```

can derive language elements (i.e., substitute definitions for abstractions):

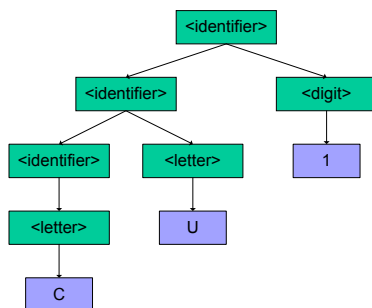
```
<identifier> → <identifier> <digit>
              → <identifier> <letter> <digit>
              → <letter> <letter> <digit>
              → C <letter> <digit>
              → CU <digit>
              → CU1
```

the above is a *leftmost* derivation (expand leftmost abstraction first)

15

Derivations vs. parse trees

```
<identifier> → <identifier> <digit>
              → <identifier> <letter> <digit>
              → <letter> <letter> <digit>
              → C <letter> <digit>
              → CU <digit>
              → CU1
```



a derivation can be represented hierarchically as a *parse tree*

- internal nodes are abstractions
- leaf nodes are terminal symbols

16

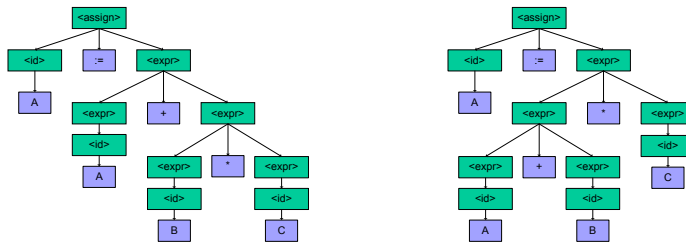
Ambiguous grammars

consider a grammar for simple assignments

```
<assign> → <id> := <expr>
<id>     → A | B | C
<expr>  → <expr> + <expr>
          | <expr> * <expr>
          | ( <expr> )
          | <id>
```

A grammar is *ambiguous* if there exist sentences with 2 or more distinct parse trees

e.g., $A := A + B * C$



17

Ambiguity is bad!

programmer perspective

- need to know how code will behave

language implementer's perspective

- need to know how the compiler/interpreter should behave

can build concepts such as operator precedence into grammars

- introduce a hierarchy of rules, lower level → higher precedence

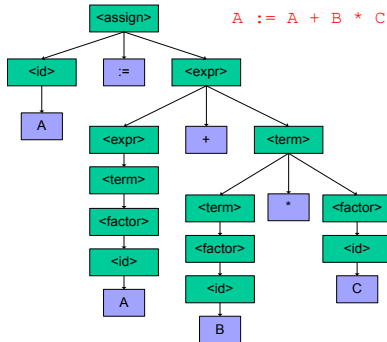
```
<assign> → <id> := <expr>
<id>     → A | B | C
<expr>  → <expr> + <term> | <term>
<term>  → <term> * <factor> | <factor>
<factor> → ( <expr> ) | <id>
```

higher precedence operators bind tighter, e.g., $A+B*C \equiv A+(B*C)$

18

Operator precedence

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$
 $\langle \text{id} \rangle \rightarrow A \mid B \mid C$
 $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle) \mid \langle \text{id} \rangle$



Note: because of hierarchy,
+ must appear above * in the parse tree

here, if tried * above, would not be able to
derive + from $\langle \text{term} \rangle$

In general, lower precedence (looser bind) will
appear above higher precedence operators in
the parse tree

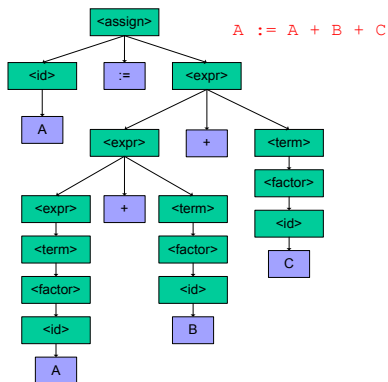
19

Operator associativity

similarly, can build in associativity

- left-recursive definitions \rightarrow left-associative
- right-recursive definitions \rightarrow right-associative

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$
 $\langle \text{id} \rangle \rightarrow A \mid B \mid C$
 $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\quad \quad \quad \mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \quad \quad \mid \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle) \mid \langle \text{id} \rangle$



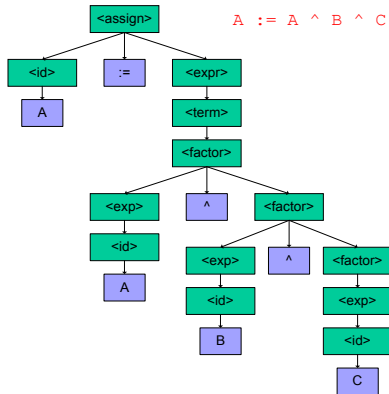
20

Right associativity

suppose we wanted exponentiation $^$ to be right-associative

- need to add right-recursive level to the grammar hierarchy

```
<assign> → <id> := <expr>
<id>     → A | B | C
<expr>  → <expr> + <term>
          | <term>
<term>  → <term> * <factor>
          | <factor>
<factor> → <exp> ^ <factor>
          | <exp>
<exp>   → ( <expr> ) | <id>
```



21

In ALGOL 60...

```
<math expr> → <simple math>
              | <if clause> <simple math> else <math expr>
<if clause>  → if <boolean expr> then
<simple math> → <term>
              | <add op> <term>
              | <simple math> <add op> <term>

<term>      → <factor> | <term> <mult op> <factor>
<factor>    → <primary> | <factor> † <primary>

<add op>    → + | -
<mult op>   → x | / | %
<primary>   → <unsigned number> | <variable>
              | <function designator> | ( <math expr> )
```

precedence? associativity?

22

Dangling else

consider the C++ grammar rule:

```
<selection stmt> → if ( <expr> ) <stmt>  
                  | if ( <expr> ) <stmt> else <stmt>
```

potential problems?

```
if ( x > 0 )  
if ( x > 100 )  
cout << "foo" << endl;  
else  
cout << "bar" << endl;
```

ambiguity!

- to which 'if' does the 'else' belong?

in C++, ambiguity remains in the grammar rules

- is clarified in the English description
(else matches nearest if)

23

Dangling else in ALGOL 60?

```
<stmt>           → <uncond stmt> | <cond stmt> | <for stmt>  
<uncond stmt>  → <basic stmt> | <compound stmt>  
<compound stmt> → begin <stmt sequence> end  
<cond stmt>    → <if stmt>  
                  | <if stmt> else <stmt>  
                  | <if clause> <for stmt>  
<if stmt>      → <if clause> <uncond stmt>  
<if clause>    → if <boolean expr> then
```

```
if x > y then  
if y > z then  
  printstring("foo");  
else  
  printstring("bar");
```

is this legal in ALGOL 60?

ambiguous?

24

Extended BNF (EBNF)

extensions have been introduced to increase ease of expression

- brackets denote optional features

```
<writeln> → writeln [ <item list> ]
```

- braces denote arbitrary # of repetitions (including 0)

```
<ident list> → <identifier> { , <identifier> }
```

- (|) denotes optional sub-expressions

```
<for stmt> → for <var> := <expr> (to | downto) <expr> do <stmt>
```

Note: could express these in BNF, but not as easily

25

BNF vs. syntax graphs

see [BNF Web Club](#) for various language grammars

- each grammar rule for a language is indexed
- in addition to BNF, syntax graphs are given



- note simplicity of LISP

26

Syntax & parsing

grammars/syntax graphs are utilized by compiler/ writers

- before compiling/interpreting, must parse the language elements

grammars/syntax graphs provide:

1. clear and concise syntax descriptions
2. can be used as the basis for a parser
3. implementations tend to be easy to maintain due to clear modularity

parsers can be top-down or bottom-up

- *top-down* parsers build the parse tree from the root (top-level abstraction) down to the leaves (terminal symbols)
e.g., recursive descent (LL) – simple, but limited (e.g., no left recursion)
- *bottom-up* parsers build the parse tree from the leaves (terminal symbols) up to the root (top-level abstraction)
e.g., shift-reduce (LR)– implemented as a PDA, more complex but more general

27

Semantics

generally much trickier than syntax

3 common approaches

- *operational semantics*: describe meaning of a program by executing it on a machine (either real or abstract)

```
Pascal code  
for i := first to last do  
begin  
  ...  
end
```

```
Operational semantics  
i = first  
loop: if i > last goto out  
  ...  
  i = i + 1  
  goto loop  
out:  ...
```

- *axiomatic semantics*: describe meaning using assertions about conditions, can prove properties of program using formal logic

```
Pascal code  
while (x > y) do  
begin  
  ...  
end
```

```
Axiomatic semantics  
while (x > y) do  
begin  
  ASSERT: x > y  
  ...  
end  
ASSERT: x <= y
```

- *denotational semantics*: describe meaning by constructing a detailed mathematical model of each language entity – PRECISE, BUT VERY EXACTING!

28