CSC 533: Organization of Programming Languages
Spring 2005

Background
- machine → assembly → high-level languages
- software development methodologies
- key languages

Syntax
- grammars, BNF
- derivation trees, parsing
- EBNF, syntax graphs
- parsing

Semantics
- operational, axiomatic, denotational

Evolution of programming

first computers (e.g., ENIAC) were not programmable
- had to be rewired/reconfigured for different computations

late 40’s / early 50’s: coded directly in machine language
- extremely tedious and error prone
- machine specific
- used numeric codes, absolute
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

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)
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 (???)

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
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)

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

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
;;; 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")
```

7
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

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
**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

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

---

**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

<table>
<thead>
<tr>
<th>Domain</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>business applications</td>
<td>COBOL</td>
</tr>
<tr>
<td>artificial intelligence</td>
<td>LISP/Scheme or Prolog</td>
</tr>
<tr>
<td>systems programming</td>
<td>C</td>
</tr>
<tr>
<td>software engineering</td>
<td>C++ or Java or Smalltalk</td>
</tr>
<tr>
<td>Web development</td>
<td>Java or JavaScript or VBScript or perl</td>
</tr>
</tbody>
</table>
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

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

  \[ \text{<ASSIGN>} \rightarrow \text{<VAR>} := \text{<EXPRESSION>} \]

- rules can be conditional using ‘|’ to represent OR

  \[ \text{<IF-STMT>} \rightarrow \text{if <LOGIC-EXPR> then <STMT>} \\
  | \text{if <LOGIC-EXPR> then <STMT>} else <STMT> \]

- arbitrarily long expressions can be defined using recursion

  \[ \text{<IDENT-LIST>} \rightarrow \text{<IDENTIFIER>} \\
  | \text{<IDENTIFIER> , <IDENT-LIST>} \]
Deriving expressions from a grammar

from ALGOL 60:

\[<\text{letter}> \rightarrow a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z\]
\[<\text{digit}> \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9\]
\[<\text{identifier}> \rightarrow <\text{letter}>\]
\[\mid <\text{identifier}> <\text{letter}>\]
\[\mid <\text{identifier}> <\text{digit}>\]

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

\[<\text{identifier}> \rightarrow <\text{identifier}> <\text{digit}>\]
\[\rightarrow <\text{identifier}> <\text{letter}> <\text{digit}>\]
\[\rightarrow <\text{letter}> <\text{letter}> <\text{digit}>\]
\[\rightarrow C <\text{letter}> <\text{digit}>\]
\[\rightarrow CU <\text{digit}>\]
\[\rightarrow CU1\]

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

---

Derivations vs. parse trees

\[<\text{identifier}> \rightarrow <\text{identifier}> <\text{digit}>\]
\[\rightarrow <\text{identifier}> <\text{letter}> <\text{digit}>\]
\[\rightarrow <\text{letter}> <\text{letter}> <\text{digit}>\]
\[\rightarrow C <\text{letter}> <\text{digit}>\]
\[\rightarrow CU <\text{digit}>\]
\[\rightarrow CU1\]

a derivation can be represented hierarchically as a parse tree

- internal nodes are abstractions
- leaf nodes are terminal symbols
Ambiguous grammars

consider a grammar for simple assignments

\[
\begin{align*}
\text{<assign>} & \rightarrow \text{id} := \text{expr} \\
\text{id} & \rightarrow \text{A | B | C} \\
\text{expr} & \rightarrow \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid (\text{expr}) \mid \text{id}
\end{align*}
\]

A grammar is ambiguous if there exist sentences with 2 or more distinct parse trees
e.g., \[ A := A + B * C \]

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 \( \rightarrow \) higher precedence

\[
\begin{align*}
\text{<assign>} & \rightarrow \text{id} := \text{expr} \\
\text{id} & \rightarrow \text{A | B | C} \\
\text{expr} & \rightarrow \text{expr} + \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} * \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow (\text{expr}) \mid \text{id}
\end{align*}
\]

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

\[
\begin{align*}
<\text{assign}> &\rightarrow <\text{id}> := <\text{expr}> \\
<\text{id}> &\rightarrow A \mid B \mid C \\
<\text{expr}> &\rightarrow <\text{expr}> + <\text{term}> \mid <\text{term}> \\
<\text{term}> &\rightarrow <\text{term}> * <\text{factor}> \mid <\text{factor}> \\
<\text{factor}> &\rightarrow ( <\text{expr}> ) \mid <\text{id}>
\end{align*}
\]

A := A + B * C

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

here, if tried * above, would not be able to 
derive + from <term>

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

Operator associativity

similarly, can build in associativity

- left-recursive definitions \(\Rightarrow\) left-associative
- right-recursive definitions \(\Rightarrow\) right-associative

A := A + B + C
Right associativity

suppose we wanted exponentiation ^ to be right-associative

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

In ALGOL 60...

|<math expr>  | <simple math>  |
|<if clause>  | <simple math> / <math expr>  |
|<simple math> | <term>  |
|<add op>    | <term>  |
|<primary>   | <add op> <term>  |

|<term>    | <factor>  |
|<primary> | <factor>  |

|<add op> | + / -  |
|<mult op> | * / %  |
|<primary> | <unsigned number>  |
|<variable> | <function designator>  |
|<math expr> | ( <math expr> )  |
Dangling else

consider the C++ grammar rule:

\[
\text{<selection stmt>} \rightarrow \text{if ( <expr> ) <stmt>}
| \text{if ( <expr> ) <stmt> else <stmt>}
\]

potential problems?

```c++
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)

Dangling else in ALGOL 60?

\[
\text{<stmt>} \rightarrow \text{<uncond stmt>} | \text{<cond stmt>} | \text{<for stmt>}
\]

\[
\text{<uncond stmt>} \rightarrow \text{<basic stmt>} | \text{<compound stmt>}
\]

\[
\text{<compound stmt>} \rightarrow \text{begin <stmt sequence> end}
\]

\[
\text{<cond stmt>} \rightarrow \text{<if stmt>}
| \text{<if stmt> else <stmt>}
| \text{<if clause> <for stmt>}
\]

\[
\text{<if stmt>} \rightarrow \text{<if clause> <uncond stmt>}
\]

\[
\text{<if clause>} \rightarrow \text{if <boolean expr> then}
\]

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

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
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

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
```pascal
for i := first to last do
  begin
    i = first
    ... ...
  end
  goto loop
out:
```

Operational semantics
```plaintext
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
```pascal
while x > y do
  begin
    ASSERT: x > y
    ... ...
  end
  ASSERT: x <= y
```

Axiomatic semantics
```plaintext
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!