G22.2210

Programming Languages: PL I

B. Mishra
New York University.

Lecture # 3
September 25 1995

—Slide 1—

Formal Syntax

- Syntax and semantics of a language is described by a **meta-language**
- **Abstract Syntax** lists all possible forms for each of the syntactic classes. It lists the syntactic classes along with the symbols that stand for arbitrary elements of the classes.
- **Concrete Syntax** tells us the *phrase structure* of a well-formed sentence in the grammar.
- **Example** Abstract syntax tells us that following statements are well-formed

```
if a then p
if a then p else q
if a then if b then p else q
```

Concrete syntax tells us which `then` the `else`-part matches to.

—Slide 2—

Meta-Languages for Concrete Syntax

- **Variants of CFG**
  (Context-Free Grammar)
  - BNF (Backus-Naur Form)
    (Also called PBF—Panini-Backus Form)
  - EBNF (Extended BNF)
  - Syntax Diagrams (or Syntax Charts)

—Slide 3—

Backus-Naur Formalisms

- **Terminal Symbols (Tokens):**
  Atomic Symbols in a language: a, b, . . ., 1, 2, . . ., +, *, . . ., or, div, . . .
- **Non-terminal Symbols (Syntactic Constructs):**
  <expression>, <term>, <literal>, . . .
- **Starting Nonterminal:**
  A distinguished non-terminal representing the main-construct.
- **Production Rules:**
  Rules specifying components of a construct.
  1. LHS = Non-terminal
  2. RHS = A string of terminals & non terminals

```
LHS ::= RHS
<expression> ::= <term>
               | <expression> <addop> <term>
<term> ::= <factor> | <term> <multop> <factor>
```

—Slide 4—

Example: Grammar for Expression

```
<expression> ::= <term>
          | <expression> <addop> <term>

<term> ::= <factor>
          | <term> <multop> <factor>

<factor> ::= <identifier>
          | <literal>
          | <expression>

<identifier> ::= a | b | c | ... | z

<literal> ::= 1 | 2 | 3 | ... | 9

<addop> ::= + | - | or

<multop> ::= * | / | div | mod | and
```

—Slide 5—

Phrase Structure

- The grammar specifies the phrase structure (i.e., expression, terms, factors, etc.) of the valid text...Not merely what text is recognizable.
- The phrase structure disambiguates in the following example, by assigning *higher precedence* to `<multop>` than `<addop>`.

```
 a + b*c                     a*b + c
      <expression><addop><term>
```

- The phrase structure also disambiguates by making both `<multop>` and `<addop>` *associate* to the left.

```
 a / b / c                a - b - c
<term><multop><factor>    <expression><addop><term>
```

- In general, there is no way of testing whether a syntax is ambiguous.

—Slide 6—

Extended BNF

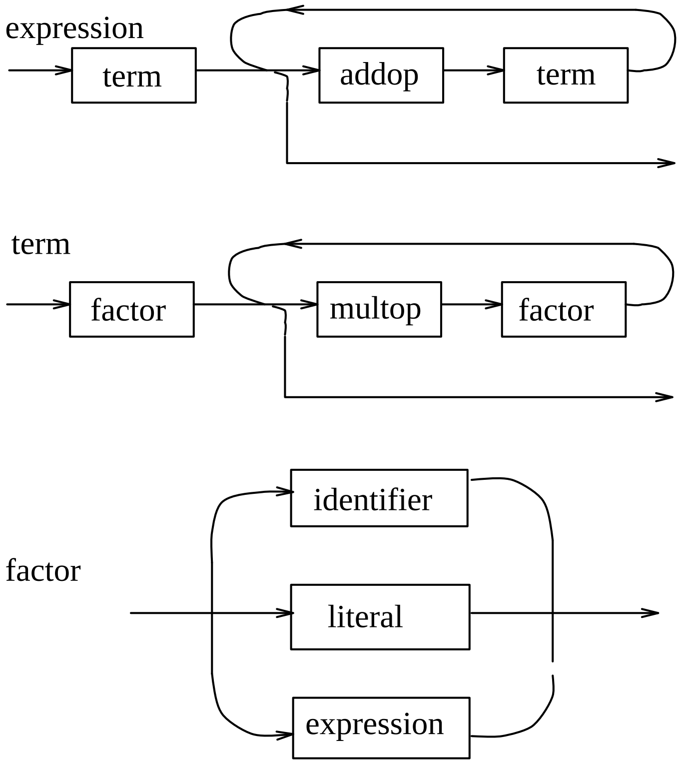
- **Nonterminals** begin with capital letters
- **Terminals** are quoted
- **Metalanguage**
  `...|...` = choice, `(...)` = grouping
  `{...}` = repetition (zero or more)
  `[...]` = optional construct
- *Example*

```
<expression> ::= <term> {<addop> <term>}
<term> ::= <factor> {<multop> <factor>}
<factor> ::= <identifier> | <literal> | <expression>
<identifier> ::= a | b | c | ... | z
<literal> ::= 1 | 2 | 3 | ... | 9
<addop> ::= + | - | or
<multop> ::= * | / | div | mod | and
```

—Slide 7—

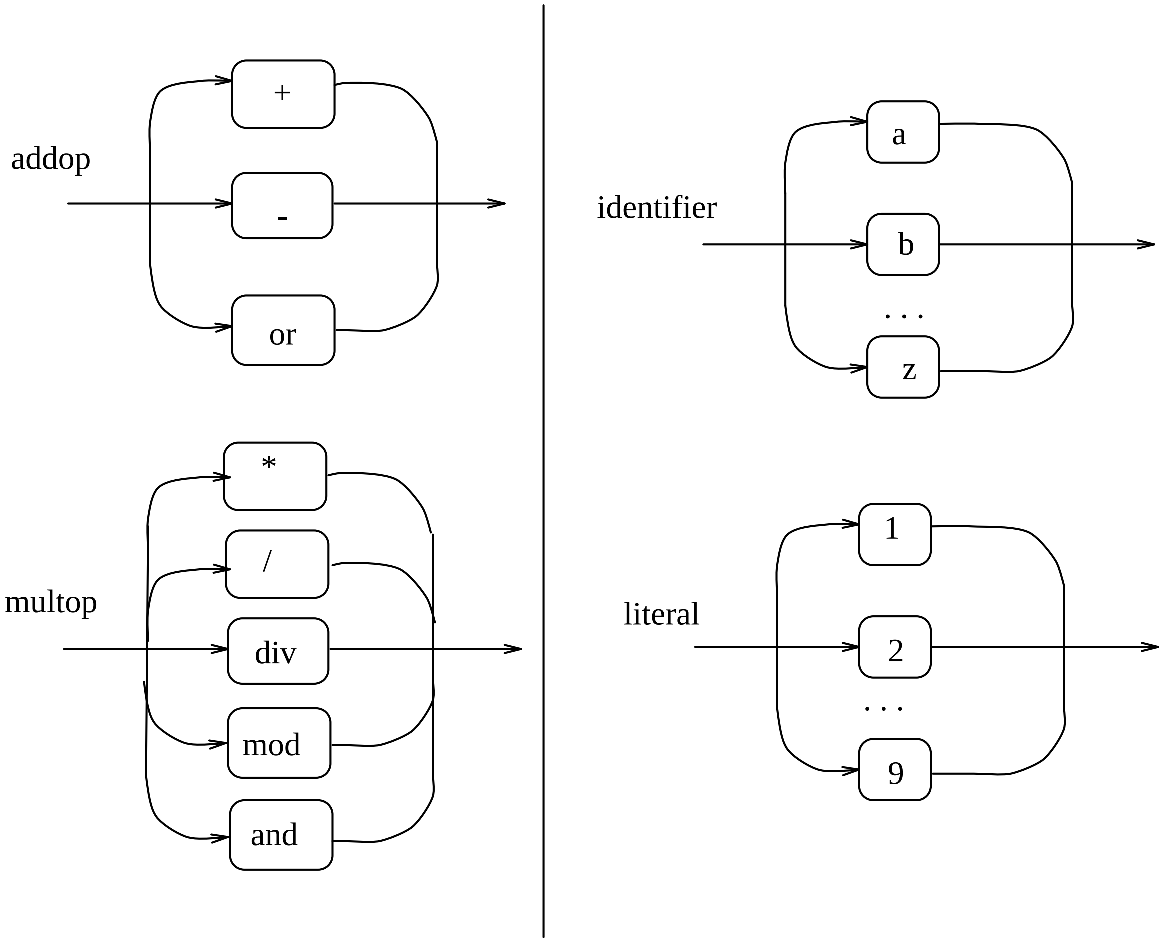
Syntax Charts

- A graphical way of writing the productions (grammar)
- **Nonterminal** ⇒ Sub-Charts
- **Productions** ⇒ Paths through the charts



—Slide 8—

Syntax Charts (contd)



—Slide 9—

Formal Semantics

- A formal description of the *semantics* of a programming language is a precise specification of the meaning of programs.
- To be used by
  Programmers, **Language Designers & Implementers**, **Theoreticians**—investigating language properties.
- **Denotational Semantics**
  A Semantic Function: mapping Syntactic Structures into Mathematical Objects.
  Denotational: Meaning of any composite phrase is expressed in terms of the meanings of its immediate constituents.

—Slide 10—

Example: Binary Numerals

- **Abstract Syntax**
  $N \in \mathbf{Nml}$ Binary Numerals

```
N ::= 0 | 1 | N0 | N1
```

- **Semantic Domain**

$$\begin{aligned} \mathbf{N} &= \{\text{zero}\} + \mathbf{N} \qquad \text{natural numbers} \\ &= \{0, 1, 2, \ldots\} \end{aligned}$$

- **Semantic Function**

$$\mathcal{N} \;:\; \mathbf{Nml} \to \mathbf{N}$$

$$\begin{aligned} \mathcal{N}[[0]] &= 0 \\ \mathcal{N}[[1]] &= 1 \\ \mathcal{N}[[N0]] &= 2 \times \mathcal{N}[[N]] \\ \mathcal{N}[[N1]] &= 2 \times \mathcal{N}[[N]] + 1 \end{aligned}$$

—Slide 11—

Semantic Domains

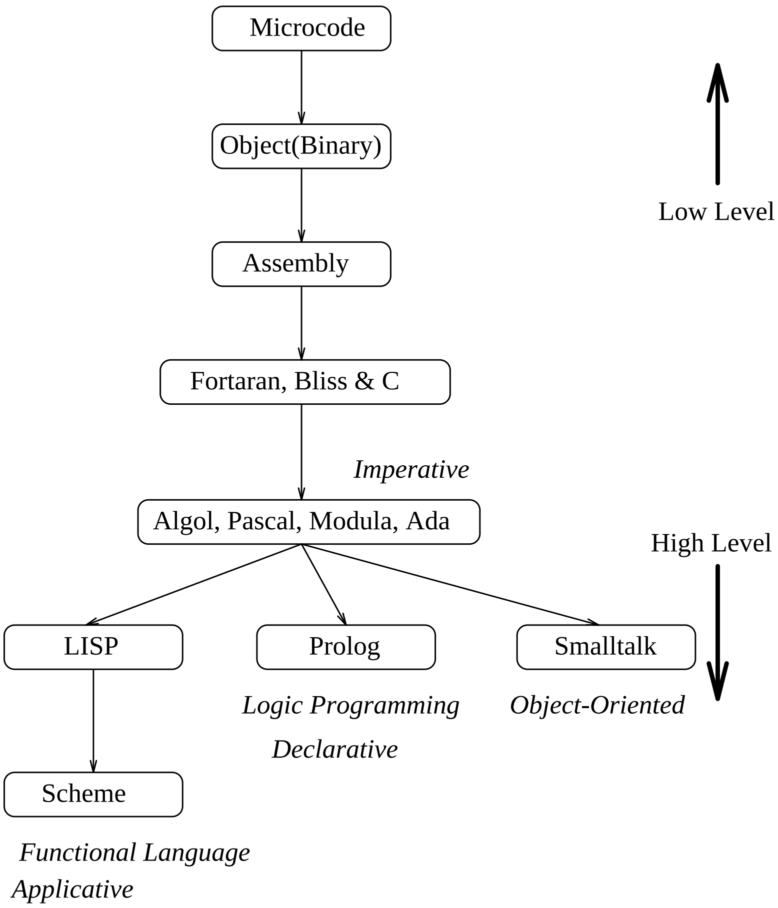
- **Basic Values**, **B**: E.g., Truth Values, Integers, ...
- **Stores**, $s \in \mathbf{S} = \mathbf{Ide} \rightarrow (\mathbf{B} + \{\text{unused}\})$
  Maps an identifier $I \in \mathbf{Ide}$ to its value $s[[I]]$.
- **Expressions**, $E \in \mathcal{E} : \mathbf{Exp} \rightarrow \mathbf{S} \rightarrow (\mathbf{B} + \{\text{error}\})$
  $\mathcal{E}[[E]]s$ is the value of E relative to store s.
- **Commands**, $C \in \mathcal{C} : \mathbf{Com} \rightarrow \mathbf{S} \rightarrow (\mathbf{S} + \{\text{error}\})$
  $\mathcal{C}[[C]]s$ is the result of executing C relative to store s.
- Thus, we have *semantics of* ;

$$\mathcal{C}[[C_1; C_2]]s = \begin{cases} \mathcal{C}[[C_2]]g, & \text{if } g \in \mathbf{S} \\ & \text{where } g = \mathcal{C}[[C_1]]s \\ \text{error}, & \text{otherwise.} \end{cases}$$

- Other domains: **Environments**, **Continuations**, ...

—Slide 12—

Hierarchy of Languages



- "Low Level" means close to machine language
- "High Level" means away from machine language —Closer to natural description of an algorithm.

—Slide 13—

Classes of Language

- **Imperative** (procedural), **Applicative** (functional) & **Declarative**
- **Imperative**:
  - A program is a sequence of commands changing state/store.
  - Objects are constructed by these statements.
- **Applicative**:
  - A program is a sequence of function definitions and applications.
- **Declarative**:
  - Objects are described rather than constructed.

—Slide 14—

Imperative Language: Assignment

- Objects in a program have two attributes:
  - *Location*: L-value
  - *Value*: R-value

```
X := X + 1;          (* Pascal *)
X := .X + 1;         (* BLISS *)
X := !X + 1;         (* ML *)
```

- In Pascal, `X` in LHS is the L-value of `X` and `X` in RHS is the R-value of `X`.
- In BLISS, `X` refers to the L-value and `.X`, the R-value. BLISS allows arithmetic on L-values (pointer arithmetic). `.` is an explicit *dereferencing operator* in BLISS.
- In ML, `X` refers to the L-value and `!X`, the R-value. `!` is an explicit *dereferencing operator* in ML.

—Slide 15—

L-values

- Some languages allow L-valued expression: E.g., in `C++`:

```
int a[10];
int& f(int i){return (a[i]);}
f(5) = 17;
```

  `f` is an L-valued function

- If two L-valued expressions denote the same location, they are called *aliases* for that location.
- FORTRAN allows explicit aliasing via `EQUIVALENCE` construct.
- Every L-valued expression has an R-value; but *not the converse.*

—Slide 16—

Variations on Assignment

- **Update Operation**

```
L +:= E;                (* ALGOL 68 *)
L += E;                 /* C        */
```

L-value of `L` contains the sum of R-values of `L` and `E`. R-value of `L` is obtained from a single evaluation of its L-value.

- **Multiple Targets**:

```
L1 := L2 := ... := Ln := E;
            (* ALOGOL 60 *)
```

All the L-values of L-expressions `L1`, `L2`, . . ., `Ln`, and the R-value of `E` are evaluated. Then all the L-values are updated.

—Slide 17—

Variations on Assignment (contd)

- **Multiple Assignments**:

```
L1, L2, ..., Ln := E1, E2, ..., En;
         (* ALOGOL 60 *)
```

  All the L-values of L-expressions `L1`, `L2`, ..., `Ln`, and then all the R-values of `E1`, `E2`, ..., `En` are evaluated. Then all the L-values are updated, while maintaining positional correspondence.

- **Assignment Expression**:

```
L := E                        (* ALOGOL 68 *)
L = E                         /* C         */
```

  The expression's value is the R-value of `L`. The expression updates `L` as a side effect.

```
if((n += a) > 0) n--;   /* C */
a = b = c = d = e;
/* C, = is right-associative */
```

—Slide 18—

Pointer

- **Pointer** (Access, Anonymous Variable)
  A variable `E` whose R-value is an L-valued expression (or a special value `null`). Its L-value is a location giving access to a storage indirectly. The L-value of `E^` is an anonymous variable which can be updated as any other L-valued expression

  ```
  E^ := E^ + 1;
  ```

- **Allocation & Disposal**
  As pointers allow storage to be addressed indirectly, it may be allocated and disposed of at arbitrary execution points.

  In some languages, allocated locations are subsequently disposed of explicitly by the user. In others, inaccessible locations are automatically searched for and disposed—*garbage collection*.

- **Inaccessible Locations**

  ```
  new(p); p := nil;
  ```

—Slide 19—

Storage Insecurities

- **Storage Insecurities**
  Dangling reference *is a pointer to a location that has potentially been used for another purpose—*

  Extent (lifetime) of the location ended before all ways of accessing the location have ended.
- Dangling reference can be created by *aliasing* and by *implicit release of storage in an activation record*, with a pointer pointing to that storage.

```
var p, q: ^integer;
begin
  new(p);
  q := p;
  dispose(p)
end;
```

```
var p: ^integer;
procedure q;
  var i: integer;
begin
  p := ADDR(i)
end;
```

—Slide 20—

Binding

- **Binding**
  Association of a *name* to an *attribute*. Following are examples of some of the attributes:
  - *L-value*—Location.
  - *Type*—The set of possible R-values allowing a set of allowable operations on them.
  - *Miscellaneous Constraints*—Assertions, array bounds, discriminant or tag values.
- **Binding Points**:
  Binding is done by *declarations* & happens at different and invisible points after the program is submitted for execution.

  The later the binding, the more flexible is the language.

—Slide 21—

Referential Transparency

- **Scope**

  Scope of a name or a declaration =

  The section of a program text in which the name has the attributes established by declaration

- *Referential transparency.*

  Thus scope corresponds to a *local name space*. Bound occurrences of a variable can be renamed without changing the meaning.

```
function succ(x:integer):integer;
begin
  succ := x + 1;
end;
```

```
function succ(y:integer):integer;
begin
  succ := y + 1;
end;
```

—Slide 22—

Types

- **Static Type-Checking**

  The type of an expression is known at the *compile time*.

  A language is **strongly-typed**, if all type checking can be done at compile time.

  —Fewer programming errors

  —Better compiled code.

- A language is **type complete** if all the objects in the language have equal status (first class citizens).

- *Type Insecurities*

  A domain incompatibility cannot be determined at compile time.

- Domain incompatibility is handled either by invoking an *exception* that aborts the program with an error message or by *type coercion*.

—Slide 23—

Type Insecurities & Coercion

- Example

```
var                              (* PASCAL *)
  wide:1..100; narrow:10..20; farout:150..300;
begin
  narrow:=farout; wide:=narrow; narrow:=wide
end;
```

–Compiler cannot determine whether the last assignment is illegal.
–Ada solves this by assigning a new compile-time *type* for every subrange constraint.

- *Type Coercion*

  If the operation and its arguments are incompatible then convert the *argument* or the *operation*, so that the types are compatible.

```
var                              (* PASCAL *)
  x: real; i: integer;
x := i;
```

—Slide 24—

Type Equivalence

- In the presence of structured types and user-defined types, it is necessary to determine if two types are equivalent.
- Two categories:
  - *Name Equivalence*: Types with same name.
  - *Structural Equivalence*:
    Types with same structure.
- Example

```
declare
  type BLACK is INTEGER;
  type WHITE is INTEGER;
  B:BLACK; W:WHITE; I:INTEGER;
begin W := 5; B := W; I := B; end;
```

–All assignments are legal under structural equivalence;
–All assignments are illegal under name equivalence.

—Last Slide—

Type Equivalence (Contd)

- Structural equivalence is hard to determine:

```
--Ada
type T1 is record          type T2 is record
  X:INTEGER;                 X:INTEGER;
  N:access T1                N:access T2
end record;                end record;

type T3 is record          type T4 is record
  X:INTEGER;                 X:INTEGER;
  N:access T2                N:access record
end record;                     X: INTEGER;
                                N: access T4
                             end record;
                           end record;
```

- **Examples**

`C`	*Structural Equivalence*
`C++`	*Name Equivalence*
PASCAL	*Declaration/Name Equivalence*
Ada	*Name Equivalence*

[End of Lecture #3]