# Grammar Rules in Prolog

# Backus-Naur Form (BNF)

- BNF is a common grammar used to define programming languages
  - » **Developed in the late 1950's**

- Because grammars are used to describe a language they are said to produce **sentences**

# Grammars and Design

◊ Grammars can be used to describe the structure of objects and computations.

» **Can be used to describe the structure of input**

> **Parse**

» **Can be used to generate output**

> **Compute**

» **Can be used to describe the structure of algorithms**

> **Design**

# Grammar Definition

◊ A grammar, **G**, is a 4-tuple **G = <T, N, S, P>,** where

» **T – a set of terminal symbols**

> **They represent themselves**

– **A, begin, 123**

» **N – a set of non-terminal symbols**

> **They are enclosed between '<' and '>'**

– **<program> <while> <letter> <digit>**

» **S ∈ N – the starting symbol**

# Grammar Definition – 2

» P – is a finite set of production or rewrite rules of the form

$$\alpha ::= \beta$$

> $\alpha$ and $\beta$ are sequences, strings, of terminal and non-terminal symbols

> $| \alpha | \geq 1$

> $\alpha$ contains at least one non-terminal symbol

# Types of Grammars

- ◊ Type 0 – Unrestricted or General grammars
  - » **Correspond to Turing machines**
  - » **Can compute anything**

- ◊ Type 1 – Context sensitive grammars
  - » **In general not used, as they are too complex**

- ◊ Type 2 – Context free grammars
  - » **Often used to describe the structure of programming languages**

# Types of Grammars – 2

◊ Type 3 – Regular grammars
  » **Correspond**
    > **Regular expressions**
    > **Finite state machines**

  » **Most business problems can be described with regular grammars**
    > **Although context free grammars are used, due to their ease of use**

# Unrestricted Grammar

- No restrictions on the definition
  - **In particular permits $|\beta| < |\alpha|$**
    - **Permits erasure of terminal symbols**

# Context Sensitive Grammar

◊ Restrict productions such that there is no erasure

» $|\beta| \geq |\alpha|$

> One exception is that the starting symbol may be in the production <Start> ::= $\varepsilon$ to be able to produce the empty sentence

◊ The following defines the language

$A^n B^n C^n$ for $n \geq 1$

(1) <S> ::= <A> <B> C
(2) <S> ::= <A> <B> <S> C
(3) <B> <A> ::= <A> <B>
(4) <B> C ::= B C
(5) <B> B ::= B B
(6) <A> B ::= A B
(7) <A> A ::= A A

# Context Free Grammar

◊ Restrict $\alpha$ to be a single non-terminal

» **$|\alpha| = 1$**

> **This permits non-terminals to be removed**

– **Note there is no erasure as terminals cannot be removed**

◊ The following defines the language

**$A^n B^n$ for $n \geq 0$**

(1) **<S> ::= $\varepsilon$**

(2) **<S> ::= A <S> B**

# Regular Grammar

◊ Restrict $\alpha$ to be a single non-terminal

◊ Restrict $\beta$ to have at most one non-terminal, with the non-terminal, if it occurs, being at either end of $\beta$

» **$|\beta| \geq 1$**

> **One exception is that the starting symbol may be in the production <Start> ::= $\varepsilon$ to be able to produce the empty sentence**

◊ Can restrict, without loss of generality to productions of the following structure giving a **Right Regular Grammar**

(1) **<non terminal> ::= terminal**
(2) **<non terminal> ::= terminal <non terminal>**

# Sentence Generation for $A^n$ $B^n$

| | |
|---|---|
| ◊ <S> → $\varepsilon$ | Rule 1 |
| ◊ <S> → A <S> B | Rule 2 |
| → A B | Rule 1 |
| ◊ <S> → A <S> B | Rule 2 |
| → A A <S> B B | Rule 2 |
| → A A B B | Rule 1 |
| ◊ <S> → A <S> B | Rule 2 |
| → A A <S> B B | Rule 2 |
| → A A A <S> B B B | Rule 2 |
| → A A A B B B | Rule 1 |

◊ …

# Parsing & Prolog

◊ Parsing is the opposite of sentence generation
  » **Task is to find a sequence of rules that produce a given sentence**

◊ Prolog has a built-in notation for representing grammar rules called **Definitive Context Grammar (DCG)**

# Parsing & Prolog – 2

◊ In a DCG the grammar for **$A^n$ $B^n$** is represented as follows

(1) **S --> [ A ] , [ B ] .**
(2) **S --> [ A ] , S , [ B ] .**

**Upper case is used in the slide for easier reading, in Prolog lower case (constants) would be used for A and B and not upper case (variables).**

# DCG Translation

- DCG statements are translated into Prolog
- The following are examples.

```
n --> n1 , n2 , … , nn .

    n (S, Rest) :-
            n1(S, R2), n2(R2, R3) , … , nn(Rn, Rest) .

n --> [ T1 ] , [ T2 ] , … [ Tn ] .

    n([T1, T2, … , Tn l Rest] , Rest) .

n --> n1 , [ T2 ] , n3 , [ T4] .

    n(S, Rest) :- n1(S, [T2 l R3]) , n3(R3, [T4 l Rest]) .

n --> [ T1 ] , n2 , [ T3 ] , n4 .

    n([T1 l R2], Rest) :-
            n2(R2, [ T3 l R4]) , n4(R4, Rest) .
```

# Translation of $A^n$ $B^n$

S --> [ A ] , [ B ] .
S --> [ A ] , S , [ B ] .

==>

s ( [ a , b | Rest ] , Rest ) .

s ( [ a | R1 ] , Rest) :- s ( R1, [ b | Rest ] ) .

◊ Every sentence is represented by 2 lists

» **Difference lists of symbols**

> **The first list is the sentence you are parsing**

> **The second list is the part of the sentence that is left-over when parsing is done**

Sample queries

s ( [a , b], [ ] ).
s ( [a , a , b , b] , [ ] ).
s ( [a , a , b , b , c] , [c] ).

# Movement example

```
move --> step.
move --> step, move.
step --> [up].
step --> [down].
```

**Example queries**

```
move ( [up, up, down] , [ ] ).
move ( [up, up, left] , [ ] ).
move ( [up, M, up] , [ ] ).
```

**Translation**

```
move ( List , Rest ) :- step ( List , Rest ).
move ( List1 , Rest) :- step ( List1 , List2 ) , move ( List2,  Rest ).
step ( [up l Rest] , Rest ).
step ( [down l Rest] , Rest ).
```

# P is a T example using determinants

```
parse --> [ P ] , [ is , a  ] ,  [ T ] .
```

Example query

```
parse ( [ 'John' , is , a , person ,  '.' ] , [ ] ).
```

Translation

```
parse ( S , Sr) :- det1 ( S , S0 )
                 , det2 ( S0 , S1 )
                 , det3 ( S1 , S2 )
                 , det4 ( S2 , Sr ).
det1 ( [ P l St ] , St ) .
det2 ( [ is, a l St ] , St ) .
det3 ( [ T l St ] , St ) .
det4 ( [ '.' l St ] , St ) .
```

# Grammars & Algorithms

◊ Unrestricted grammars have been used to write programs

  » **Snobol language was used to develop a system called MUMPS that was used in hospital applications circa 1960's–1970's**

# SNOBOL

- In Snobol a grammar is defined to translate (rewrite) an input string of symbols to an output string of symbols
  - **The production rules are applied using the Markov algorithm**
    - **Developed during the 1940's as yet another description of what it means to compute**
  - **Works in a similar way to Prolog**
    - **Pattern matching takes place on strings, instead of compound terms**

# Markov Algorithm

- ◊ Input
  - » A numbered set of productions $\alpha \rightarrow \beta$
    - > Numbering is from 1 up
  - » An input string – maStr – over the alphabet
    - > No distinction needed for terminals and non-terminals
- ◊ Computation
  - » The productions are applied to the sequence of strings beginning with the input string
- ◊ Output
  - » The resulting string when no production is applicable

# Markov Algorithm

```
PROCEDURE
VAR j : integer              { An index to a production.}
;     k : integer            { An index to the occurrence
                               of an alpha [ j ] in maStr.}
;   notAtEnd : boolean       { Goes FALSE when algorithm is done.}

;   BEGIN
        j := 1               { Start at production 1.}
    ;   notAtEnd := true

    ;   WHILE notAtEnd DO BEGIN
            … DO loop body – see next slide
         END
  END
```

# Markov Algorithm Body of Loop

```
            { Find left most occurrence of alpha.}
  k := index ( maStr, 1 , alpha [ j ] )

; IF k = 0 THEN                {No alpha, try the next production.}
    BEGIN j := j+1             {No alpha, try the next production.}

    ;   IF j > prodCount          {Do we have a production to try?}
          THEN notAtEnd := false     {No production,  stop.}
        END
    END

  ELSE BEGIN             {Found alpha, apply production.}
      replace (maStr, beta [ j ] , k , alpha [ j ] . length )
      j := 1                 {Start with first production again.}
  END

END
```

# MA Add two binary numbers

◊ Alphabet

- » 0 1    <- The binary digits.
- » a    <- Remember a 1.
- » b    <- Remember a 0.
- » c    <- Remember a carry.
- » N    <- A 1 in the sum.
- » Z    <- A 0 in the sum.
- » X    <- Separator for the two input numbers.

# MA Add two binary numbers – 2

- Productions
  - **a1 -> 1a ; a0 -> 0a ;** **<- Travel right with a one**
  - **b1 -> 1b ; b0 -> 0b ;** **<- Travel to right with a zero**
  - **1c -> c0 ; 0c -> 1 ; c -> 1 ;** **<- Propagate a carry**
  - **1a -> cZ ; 0a -> N ; Xa -> N ;** **<- Add one to least sig digit of n2**
  - **1b -> N ; 0b -> Z ; Xb -> Z ;** **<- Add zero to least sig digit of n2**
  - **1X -> Xa ; 0X -> Xb ;** **<- Move least sig digit of n1 to add position**
  - **N -> 1 ; Z -> 0 ;** **<- Recover all zeros and ones**
- An input string
  - **101X1101**

# SNOBOL – Syntactic Sugar

◊ Some productions terminate with a period

» **If such a production is applied, the computation terminates**

◊ Some productions are labeled

◊ Some productions have success and failure tags

» **If such a production is applied, the Markov algorithm resumes from the production labeled by the success tag**

» **If such a production is not applied, then the Markov algorithm resumes from the production labeled by the failure tag**