

## Structure of Programming Languages – Lecture 3

CS 636 – 536

April 20, 2009

- 1 Syntax and its Specification
  - Syntax: Definition
  - Extended Backus-Naur Form
  - Railroad Diagrams
  - The Definition of Pascal
- 2 Parsing
  - Ad-hoc Parsing
  - Parsing Based on EBNF
- 3 The Syntax of FORTH

## Language, Metalanguage, and meta-meta language.

- We need a way to define the syntax, or grammar, of a programming language.
- This definition must be unambiguous.
- It must be suitable for processing by a computer so that we can use it to construct the front end of a compiler.
- The language described must be straightforward to parse. Formally, the grammars for programming languages all belong to the category called "context-free languages".

A **metalanguage** is a language used for defining a language. We will use English as a metalanguage to talk about EBNF, which is used as a metalanguage for defining programming languages.

## An EBNF grammar consists of:

- A set of terminal symbols, which are the keywords and syntactic markers of the language being defined.
- A set of nonterminal symbols, which correspond to the syntactic categories and kinds of statements of the language.
- One non-terminal symbol, often  $S$ , is designated as the starting symbol.
- A series of rules, called productions, that specify how each nonterminal symbol may be expanded into a phrase containing terminals and nonterminals. Every nonterminal has one production rule, which may contain alternatives.

## The syntax for EBNF itself:

- Terminal symbols will be written in **boldface** and/or enclosed in 'single quotes'.
- Nonterminal symbols will be written in non-bold type and/or enclosed in  $\langle$ angle brackets $\rangle$ .
- Production rules. The nonterminal being defined is written at the left, followed by a “::=” sign (which we will pronounce as “goes to”). After this is a set of options, which define how the nonterminal can be expanded. The rule extends up to but does not include the “.” at the end.
- When a nonterminal is *expanded* it is replaced by one of the options from its definition.
- Blank spaces between the “::=” and the “.” are ignored.

## Syntax for EBNF Production Rules:

- Alternatives are separated by vertical bars.

This indicates that an 's' may be replaced by an 'a' or a 'bc':

$$s ::= a \mid bc .$$

- Parentheses may be used to indicate grouping. For example, this indicates that an 's' may be replaced by an 'ad' or a 'bcd':

$$s ::= ( a \mid bc ) d .$$

- Something enclosed in square brackets is optional. For example, this rule says that an 's' may be replaced by an 'ad' or simply by a 'd':

$$s ::= [a] d .$$

- Zero or more repetitions of a unit is indicated by enclosing the unit in curly braces. This rule says that an 's' may be replaced by a 'd', an 'ad', an 'aad', or a string of any number of 'a's followed by a single 'd' and one or more 'b's.

$$s ::= \{a\} d b \{b\} .$$

## EBNF grammar for Gibberish.

- The starting symbol is  $S$ .
- Terminal symbols are: **A B D**
- Nonterminal symbols are:  $S$ , *stop*
- Productions:  
 $S ::= S \mathbf{A} S \mid stop$   
 $stop ::= \mathbf{B} \mid \mathbf{D}$
- Using this grammar to generate a sentence of Gibberish:
  - $S$
  - $SAS$
  - $SASAS$
  - $stop\mathbf{A}stop\mathbf{A}stop$
  - $\mathbf{B} \mathbf{A} \mathbf{B} \mathbf{A} \mathbf{D}$

# Syntax Diagrams

An alternative formal definition metalanguage was developed for Pascal; it is often called “railroad diagrams”. It has the same elements as EBNF, but they are presented in a 2D graphic format:

- Terminal symbols are **boldface** and enclosed in ovals. Nonterminal symbols are written in non-bold type.
- Production rules: the nonterminal being defined is written at the left, followed by an arrow.
- Alternatives are shown by branches in the arrow.
- To expand a non-terminal, follow some branch of the arrow to its end at the right.
- An optional element is handled by an empty arrow branching around it.
- Repetitions of a unit are shown by the arrow looping back on itself.

## The Syntax for part of Pascal.

- $program ::= \langle \text{program-heading} \rangle \text{' ;' } \langle \text{program-block} \rangle \text{' .' } .$
- $program\text{-heading} ::=$   
 $\text{' program' } \langle \text{identifier} \rangle [ \text{' (' } \langle \text{program-parameters} \rangle \text{' )' } ] .$
- $program\text{-parameters} ::= \langle \text{identifier-list} \rangle .$
- $identifier\text{-list} ::= \langle \text{identifier} \rangle \{ \text{' ,' } \langle \text{identifier} \rangle \} .$
- $program\text{-block} ::= \langle \text{block} \rangle .$
- $block ::= \langle \text{label-declaration-part} \rangle \langle \text{constant-declaration-part} \rangle$   
 $\langle \text{type-declaration-part} \rangle \langle \text{variable-declaration-part} \rangle$   
 $\langle \text{procedure-and-function-declaration-part} \rangle$   
 $\langle \text{statement-part} \rangle .$
- $variable\text{-declaration-part} ::= [ \text{' var' } \{ \langle \text{identifier-list} \rangle \text{' :' } \langle \text{typename} \rangle \text{' ;' } \} ] .$

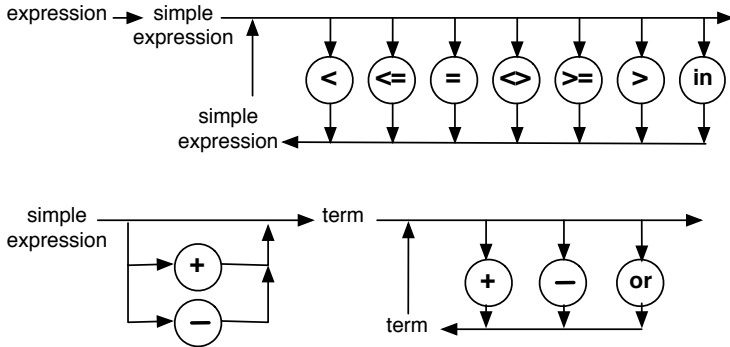
## Continuing with Pascal.

- $statement\text{-}part ::= \text{compound statement} .$
- $compound\text{-}statement ::= \text{'begin'} \langle \text{statement-sequence} \rangle \text{'end'} .$
- $statement\text{-}sequence ::= \langle \text{statement} \rangle \{ \text{';} \langle \text{statement} \rangle \} .$
- $statement ::= [ \langle \text{label} \rangle \text{' : ' } ] ( \langle \text{simple-statement} \rangle | \langle \text{structured-statement} \rangle ) .$
- $simple\text{-}statement ::= \langle \text{empty-statement} \rangle | \langle \text{assignment-statement} \rangle | \langle \text{procedure-statement} \rangle | \langle \text{goto-statement} \rangle .$
- $structured\text{-}statement ::= \langle \text{compound-statement} \rangle | \langle \text{conditional-statement} \rangle | \langle \text{repetitive-statement} \rangle | \langle \text{with-statement} \rangle .$

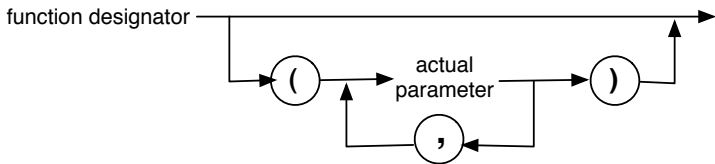
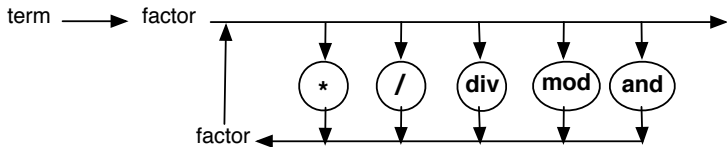
## Expressions, Precedence and Associativity in Pascal.

- $statement\text{-}part ::= \text{compound statement} .$
- $compound\text{-}statement ::= \text{'begin'} \langle \text{statement-sequence} \rangle \text{'end'} .$
- $statement\text{-}sequence ::= \langle \text{statement} \rangle \{ \text{';} \langle \text{statement} \rangle \} .$
- $statement ::= [ \langle \text{label} \rangle \text{' : ' } ] ( \langle \text{simple-statement} \rangle | \langle \text{structured-statement} \rangle ) .$
- $simple\text{-}statement ::= \langle \text{empty-statement} \rangle | \langle \text{assignment-statement} \rangle | \langle \text{procedure-statement} \rangle | \langle \text{goto-statement} \rangle .$
- $structured\text{-}statement ::= \langle \text{compound-statement} \rangle | \langle \text{conditional-statement} \rangle | \langle \text{repetitive-statement} \rangle | \langle \text{with-statement} \rangle .$

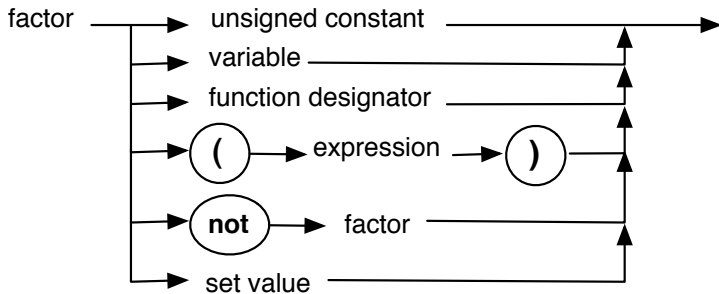
# Pascal Expressions



# Pascal Expressions



# Pascal Expressions



## Old Languages were Parsed Ad-Hoc

These comments reflect FORTRAN-IV.

- The language itself was created by collecting a lot of features. Everything about it was non-uniform and full of special cases. For example, there were half a dozen ways to punctuate a group of items. Syntax diagrams occupied 40 pages, versus 6 for Pascal.
- Everything was made more difficult because the language definition said that spaces were ignored.
- A FORTRAN-IV parser was basically hand-built. It would look at the next source-code character and try to figure out what it might be, given the current context.
- This is a famous FORTRAN parsing problem that illustrates what is wrong with ad-hoc design: `DO 200 I=1,10,2`

# Ad-Hoc Languages Today

We can list several current languages with no rhyme or reason in the design:

- The C-shell, bash, tcsh, and other UNIX shell languages.
- Perl
- $T_eX$  and  $L^A T_eX$
- http and https

These are hard to learn and hard to write correctly. They are parsed and interpreted in an ad-hoc manner. Often the semantics are complicated and hard to understand.

## Compiler compilers.

A **compiler compiler** is a program whose input is the description of the syntax and semantics of a language and whose output is a compiler. The inputs are:

- A formal definition of the language's lexical structure (expressed in EBNF).
- A formal definition of preprocessing directives, if any, and their corresponding actions.
- The EBNF definition of the language syntax, given that tokens have already been identified.
- The code to be generated for each fully-parsed nonterminal symbol in the grammar.

The compiler compiler produces the compiler that will build a parse tree (front end) and transmute the tree into the corresponding object code (back end).

## Recursive Descent Parsing: top down

- The parser starts with the starting symbol of the grammar and the beginning of the source-code file.
- It then attempts to find a match for the left end of one of the alternatives in the definition of the starting symbol.
- If the left end is found, it calls itself recursively, with the rest of the source code, to find a match for the next part of the production.
- If it fails at any point, it will backtrack and try another option from the current production.
- This process works its way through the source code and down the list of productions. It will terminate successfully when the inner, recursive calls have all terminated and a match is found for the rightmost element in the original starting production.

## Syntax Practice: Defining FORTH

Let us try to write a formal syntax to define a FORTH function.

- Starting symbol: S.
- Other nonterminals: token, ifStatement, whileLoop, countedLoop, infiniteLoop, untilLoop
- Terminal symbols: if, begin, while, repeat, until, leave, again, do, I, loop, +loop, if, else, then
- Productions: an in-class exercise to be finished at home.