A few necessary definitions

*Parse* - *vt*, to resolve (as a sentence) into component parts of speech and describe them grammatically

*Grammar* - *n*, the study of the classes of words, their inflections, and their functions and relations in the sentence

*Syntax* - *n*, the way in which words are put together to form, phrases, clauses or sentences
The Parsing Process

Syntactic Analysis (or Parsing) involves breaking a program into its syntactic components.

The Parsing Process (continued)

Nb: In the previous example,

subject phrase, predicate, adjectives, etc. were nonterminals.

definite articles, adjective, noun, verb, etc. were terminals

A language is a set of sentences formed by the set of basic symbols.

A grammar is the set of rules that govern how we determine if these sentences are part of the language or not.
The Parsing Process (continued)

The analysis is based purely on *syntax*. A syntactically correct sentence can be nonsensical:

**Example:**
A group of trout were flying east, where they hunted down camels for their dinner.

Parsing as a procedure

The parser takes tokens from scanner as necessary and produces a tree structure (or analyzes the program as if it were producing one). It is called as a procedure of the main program:

```c
struct parsenoderec *parsetree;
parsetree = parse( );
```

In most real cases, the parser actually returns a pointer to an abstract syntax tree or some other intermediate representation.
Error recovery during parsing

• The parser will (or certainly should) spot any and all syntactic errors in the program.
• This requires us to consider how we will handle recovery from any errors encountered:
  – We can consider any error fatal and point it out to the user immediately and terminate execution.
  – We can attempt to find a logical place within the program where we can resume parsing so that we can spot other potential errors as well.

Types of Parsers

• Parsers can be either top-down or bottom-up:
  – Top-down parsers build the parse-tree starting from the root building until all the tokens are associated with a leaf on the parse tree.
  – Bottom-up parsers build the parse-tree starting from the leaves, assembling the tree fragments until the parse tree is complete.
Top-down Parsers

Sentence

Subject phrase

definite article

Adjectives

noun

The

Top-down parsing assumes a certain minimum structure as we start building the parse tree.

Bottom-up parsers

Subject Phrase

def. art. adj. adj. noun

The quick brown fox

Bottom-up parsers shift by each token, reducing them into a non-terminal as the grammar requires.

Nb: Until we finish building the predicate, we have no reason to reduce anything into the nonterminal Sentence.
Types of Parsers (continued)

- Parsers can be either table-driven or handwritten:
  - Table-driven parsers perform the parsing using a driver procedure and a table containing pertinent information about the grammar. The table is usually generated by automated software tools called parser generators.
  - Handwritten parsers are hand-coded using the grammar as a guide for the various parsing procedures.

Types of Parsers (continued)

- LL(1) and LR(1) parsers are table-driven parsers which are top-down and bottom-up respectively.
- Recursive-descent parsers are top-down hand-written parsers.
- Operator-precedence parsers are bottom-up parsers which are largely handwritten for parsing expressions.
Context-Free Grammars

A context-free grammar is defined by the 4-tuple:

\[ G = (T, N, S, P) \]

where

- **T** = The set of *terminals* (e.g., the tokens returned by the scanner)
- **N** = The set of *nonterminals* (denoting structures within the language such as *DeclarationSection, Function*).
- **S** = The *start symbol* (in most instances, our program).
- **P** = The set of *productions* (rules governing how tokens are arranged into syntactic units).

Context-Free Grammars

- Context-free grammars are well-suited to programming languages because they restrict the manner in which programming construct can be used and thus simplify the process of analyzing its use in a program.
- They are called context-free because the manner in which we parse any nonterminal is independent of the other symbols surrounding it (i.e., parsing is done without respect to *context*).
- The grammars of most programming languages are explicitly context-free (although a few have one or two context-sensitive elements).
Distinction between syntax and semantics

• Syntax refers to features of sentence structure as it appears in languages.
• Semantics refers to the meaning of such structures.
• The parser will analyze the syntax of a program, not its semantics.
  – E. g., the parser does not do type-checking.
  – Semantic actions will frequently be associated with specific productions, but are not actually part of the parser.

Backus-Naur Form

BNF (Backus-Naur Form) is a metalanguage for describing a context-free grammar.

• The symbol ::= (or \rightarrow) is used for may derive.
• The symbol | separates alternative strings on the right-hand side.

Example

\[ \begin{align*}
E & ::= E + T | T \\
T & ::= T * F | F \\
F & ::= \text{id} | \text{constant} | (E)
\end{align*} \]

where E is Expression, T is Term, and F is Factor
Extended Backus-Naur Form

EBNF (Extended Backus-Naur Form) adds a few additional metasymbols whose main advantage is replacing recursion with iteration.

- \{a\} means that \(a\) is occur zero or more times.
- \([a]\) means that \(a\) appears once or not at all.

Example

Our expression grammar can become:

\[
\begin{align*}
E & ::= T \ \{ \ + \ T \} \\
T & ::= F \ \{ \ * \ F \} \\
F & ::= id \ | \ constant \ | \ (E)
\end{align*}
\]

A simple grammar

Start Symbol \[\implies\] S ::= A B c

A ::= a A | b

B ::= A b | a

The strings \textit{abbbe, aaabac, aaaaababbc} are all generated by this grammar. Can you determine how?
Another simple grammar

S ::=  a | (b S S)

Sample strings generated by this grammar include:

(b a a)  (b (b a a) a)  a

The Empty String

- Productions within a grammar can contain $\varepsilon$, the empty string.
- $A \rightarrow B$ is equivalent to $A \rightarrow B\varepsilon$
- It is also possible to write the production $A \rightarrow \varepsilon$; such productions become particularly useful in top-down parsing.
Derivations

- A derivation is a series of replacements where the nonterminal on the left of a production is replacement by a string of symbols from the right-hand side of a production.
- This may be done in one step or in many steps.

**Example**

For the grammar

\[
S ::= Aa \\
A ::= Ab | c
\]

\[
S \Rightarrow Aa \Rightarrow Aba \Rightarrow Abba \Rightarrow cbba
\]

*cbba* is ultimately derived from *S*

---

Derivations (continued)

- There are several different notations used to indicate occurs:
  - \( A \Rightarrow \alpha \) A derives \( \alpha \) in one step
  - \( A \Rightarrow^* \alpha \) A derives \( \alpha \) in zero or more steps
  - \( A \Rightarrow^+ \alpha \) A derives \( \alpha \) in one or more steps

**Example**

\[
S \Rightarrow Aa \Rightarrow Aba \Rightarrow Abba \Rightarrow cbba
\]

We can say that \( S \Rightarrow^* cbba \)
Derivations (continued)

- If the start symbol S derives a string \( \beta \) which contains nonterminals, \( \beta \) is a sentential form.
- If S derives a string \( \beta \) which contains only terminals, \( \beta \) is a sentence.

Parse Trees

A parse tree is a graphical representation of such a derivation:

```
      S
     / \  
    A   a
   / \   
  A   b
 /   /  
A b c
```
Left and right derivations

Remember our grammar:

\[
S \ ::= A \ B \ c \\
A \ ::= a \ A \mid b \\
B \ ::= A \ b \mid a
\]

How do we parse the string \textit{abbbc}?


Languages and Grammars

- A grammar is just a way of describing a language.
- There are actually an infinite number of grammars for a particular language.
- 2 grammars are equivalent if they describe the same language.
  - This becomes extremely important when parsing top-down.
  - Most programming language manuals contain a grammar in BNF or EBNF, which we may modify to fit our parsing method better.
Ambiguous grammars

- While there may be an infinite number of grammars that describe a given language, their parse trees may be very different.
- A grammar capable of producing two different parse trees for the same sentence is called **ambiguous**. Ambiguous grammars are highly undesirable.

Is it IF-THEN or IF-THEN-ELSE?

The IF-THEN-ELSE ambiguity is a classical example of an ambiguous grammar.

\[
\text{Statement ::= \ if\ Expression\ \ then\ Statement\ else\ Statement} \\
\text{\ \ |\ \ if\ Expression\ \ then\ Statement}
\]

How would you parse the following string?

\[
\text{IF } x > 0 \\
\text{ THEN IF } y > 0 \\
\text{ \ \ THEN } z := x + y \\
\text{ ELSE } z := x;
\]
Is it IF-THEN or IF-THEN-ELSE? (continued)

There are two possible parse trees:

- \[
  \text{Statement} \\
  \text{if Expression then Statement} \\
  \text{if Expression then Statement else Statement}
\]

- \[
  \text{Statement} \\
  \text{if Expression then Statement else Statement} \\
  \text{if Expression then Statement}
\]

---

\[
\begin{align*}
\text{Statement} & := \text{if Expression then Statement ElseClause} \\
\text{ElseClause} & := \text{else Statement} \mid \varepsilon
\end{align*}
\]

- \[
  \text{Statement} \\
  \text{if Expression then Statement ElseClause} \\
  \text{if Expression then Statement ElseClause else Statement}
\]

- \[
  \text{Statement} \\
  \text{if Expression then Statement ElseClause else Statement}
\]
Operator Precedence

Most programming languages have an order of precedence for operators. It would be helpful if this could be encoded into the language’s grammar.

E. g., let’s take a look at the order of precedence in Pascal:

Highest: $Unary +, Unary -, NOT$

$\ast, /, DIV, MOD, AND$

$+, -, OR$

Lowest: $=, <>, >=, <=, >, <$

Operator Precedence (continued)

This can be encoded in our grammar by considering first a production for our highest level of precedence:

$$Factor ::= Unary-operator Unary-Factor$$

$$| Unary-Factor$$

Let’s now consider the next-highest level:

$$Term ::= Term Multiplicative-operator Factor$$

$$| Factor$$
Operator Precedence (continued)

Now let’s consider the next-level:
\[ Expr. ::= Expr. \text{ Add.-op} \text{ Term} | \text{ Term} \]

And finally,
\[ Rel.-Expr. ::= Rel.-Expr \text{ Rel.-op} \text{ Expr.} | \text{ Expr.} \]

Once we add the production
\[ \text{Factor ::= Identifier} | \text{ Constant} | (\text{Rel.Expr.}) \]
we have a complete expression grammar for Pascal.

Operator Precedence (continued)

In general, we can start from the lowest order of precedence and work our way to highest in this fashion

\[ \text{ExprA ::= ExprA opA ExprB} | \text{Exprb} \]
\[ \text{ExprB ::= ExprB opB ExprC} | \text{ExprC} \]

\[ \ldots \]
\[ \text{ExprZ ::= Identifier} | \text{Const} | \ldots \]
Expression grammar in C

C was 14 levels of precedence, making its expression grammar more complex than that of most other languages:

```
Expr ::= Expr , AssnExpr | AssnExpr
AssnExpr ::= UnaryExpr AssnOp AssnExpr | CondExpr
AssnOp ::= = | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | !=
CondExpr ::= LogORExpr | LogORExpr ? Expr : CondExpr
LogORExpr ::= LogORExpr || LogANDExpr | LogANDExpr
LogANDExpr ::= LogANDExpr && InclORExpr | InclORExpr
InclORExpr ::= InclORExpr | ExclORExpr | ExclORExpr
ExclORExpr ::= ExclORExpr ^ ANDExpr | ANDExpr
ANDExpr ::= ANDExpr & EQExpr | EQExpr
EQExpr ::= EQExpr == RelExpr | EQExpr != RelExpr | RelExpr
RelExpr ::= RelExpr >= ShftExpr | RelExpr <= ShftExpr
        | RelExpr > ShftExpr | RelExpr < ShftExpr | ShftExpr
ShftExpr ::= ShftExpr >> AddExpr | ShftExpr << AddExpr
        | ShftExpr
AddExpr ::= AddExpr + MultExpr | AddExpr - MultExpr
        | MultExpr
MultExpr ::= MultExpr * CastExpr | MultExpr / CastExpr
        | MultExpr % CastExpr | CastExpr
```
Expression grammar in C (continued)

\[
\text{CastExpr} ::= (\text{typename}) \text{CastExpr} \mid \text{UnExpr}
\]

\[
\text{UnExpr} ::= \text{PostExpr} \mid ++\text{UnExpr} \mid --\text{UnExpr}
\]

\[
\mid \text{UnOp} \text{CastExpr} \mid \text{sizeof} \text{UnExpr} \mid \text{sizeof}(\text{typename})
\]

\[
\text{UnOp} ::= \& \mid \* \mid + \mid - \mid \sim \mid !
\]

\[
\text{ExprList} ::= \text{ExprList}, \text{AssnExpr} \mid \text{AssnExpr}
\]

\[
\text{PostExpr} ::= \text{PrimExpr} \mid \text{PostExpr}[\text{Expr}] \mid \text{PostExpr}(\text{ExprList})
\]

\[
\mid \text{PostExpr}.\text{id} \mid \text{Post Exp} \rightarrow \text{id} \mid \text{PostExpr} ++
\]

\[
\mid \text{PostExpr} --
\]

\[
\text{PrimExpr} ::= \text{Literal} \mid (\text{Expr}) \mid \text{id}
\]

\[
\text{Literal} ::= \text{integer-constant} \mid \text{char-constant} \mid \text{float-constant}
\]

\[
\mid \text{string-constant}
\]

JASON grammar

\[
\text{Program} ::= \text{Header DeclSec Block} .
\]

\[
\text{Header} ::= \text{program id ;}
\]

\[
\text{DeclSec} ::= \text{VarDecls ProcDecls}
\]

\[
\text{VarDecls} ::= \text{VarDecls VarDecl} \mid \text{VarDecl} \mid \varepsilon
\]

\[
\text{VarDecl} ::= \text{DataType IdList}
\]

\[
\text{DataType} ::= \text{real} \mid \text{integer}
\]

\[
\text{IdList} ::= \text{IdList}, \text{id} \mid \text{id}
\]
JASON grammar (continued)

ProcDecls ::= ProcDecls ProcDecl | ProcDecl | ε
ProcDecl ::= ProcHeader DeclSec Block ;
ProcHeader ::= procedure id ParamList ;
ParamList ::= ( ParamDecls ) | ε
ParamDecls ::= ParamDecls ; ParamDecl |

ParamDecl ::= DataType id
Block ::= begin Statements end
Statements ::= Statements ; Statement | Statement

JASON grammar (continued)

Statement ::= read id | write id
| set id = Expression
| if Condition then Statements
| ElseClause endif
| while Condition do Statements
| endwhile
| until Condition do Statements
| enduntil
| call id Arglist
| ε
JASON grammar (continued)

ElseClause ::= else Statements | ε
ArgList ::= ( Arguments ) | ε
Arguments ::= Arguments, Factor | Factor
Condition ::= Expression RelOp Expression
Expression ::= Expression AddOp Term | Term
Term ::= Term MultOp Factor | Factor
Factor ::= id | constant
RelOp ::= > | < | = | !=
AddOp ::= + | -
MultOp ::= * | /

ElseClause ::= else Statements | ε
ArgList ::= ( Arguments ) | ε
Arguments ::= Arguments, Factor | Factor
Condition ::= Expression RelOp Expression
Expression ::= Expression AddOp Term | Term
Term ::= Term MultOp Factor | Factor
Factor ::= id | constant
RelOp ::= > | < | = | !=
AddOp ::= + | -
MultOp ::= * | /