Compiler Construction

Week 03- Context-Free Grammars

A few necessary definitions

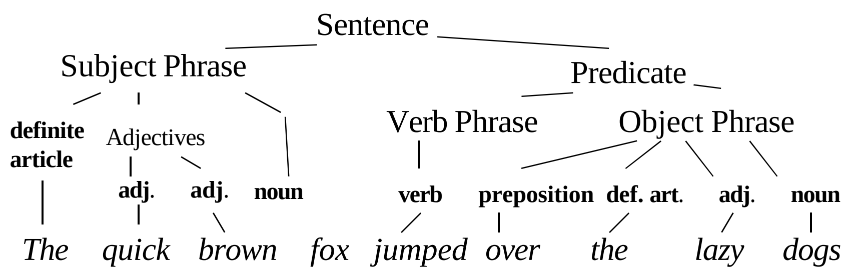
<u>*Parse*</u> - *vt*, to resolve (as a sentence) into component parts of speech and describe them grammatically

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

<u>*Syntax*</u> - *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 article, adjective, noun, verb, etc. were *terminals*

A language is a set of sentences formed 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:

```
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.

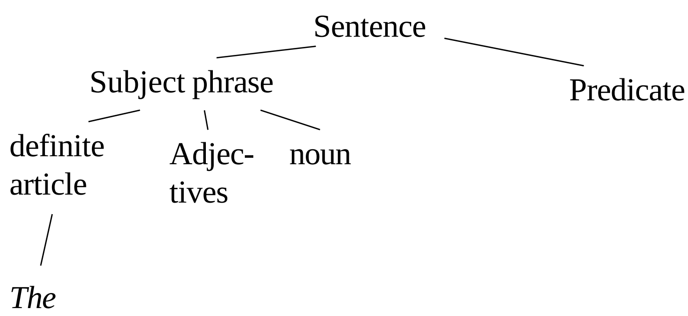
Error recovery during parsing

- 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

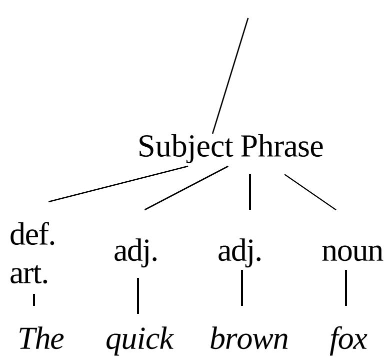
- Parsers can be either ***top-down*** or ***bottom-up***:
  - Top-down parsers build the parse-tree starting from the root building until all the tokens are associated with a leaf on the parse tree.
  - Bottom-up parsers build the parse-tree starting from the leaves, assembling the tree fragments until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

Top-down Parsers



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

Bottom-up parsers



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 (***B***ackus-***N***aur ***F***orm) 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

E ::= E + T | T
T ::= T * F | F
F ::= id | constant | (E)

where E is ***Expression***, T is ***Term***, and F is ***Factor***

A simple grammar

Start Symbol ⇨ S ::= A B c

A ::= a A | b

B ::= A b | a

The strings ***abbbc***, ***aaabac***, ***aaaababbc*** 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 ε, 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 ⇨ Aa ⇨ Aba ⇨ Abba ⇨ cbba

cbba is ultimately derived from ***S***

Derivations (continued)

- There are several different notations used to indicate occurs:

  $A \Rightarrow \alpha$ A derives α in one step

  $A \Rightarrow^{*} \alpha$ A derives α in zero or more steps

  $A \Rightarrow^{\dagger} \alpha$ A derives α in one or more steps

- Example

  $S \Rightarrow Aa \Rightarrow Aba \Rightarrow Abba \Rightarrow cbba$

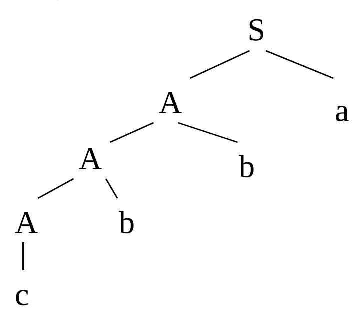
  We can say that $S \Rightarrow^{*} cbba$

Derivations (continued)

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

Parse Trees

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



Abstract Syntax Tree

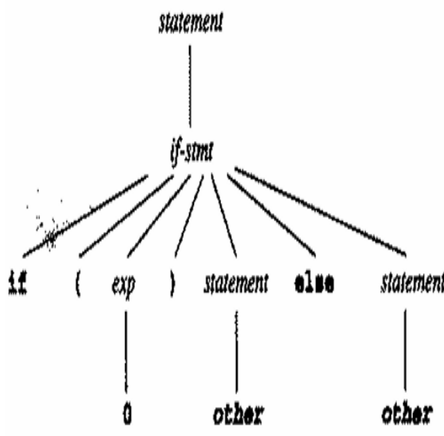
- An abstract syntax tree represents abstractions of actual source code token sequences, and token sequences cannot be recovered from them (unlike parse trees).
- Abstract syntax trees is a tree representation of a shorthand notation called abstract syntax.

Parse Tree vs Abstract Syntax Tree

statement → *if-stmt* | **other**
if-stmt → **if (*exp*)** *statement*
| **if (*exp*)** *statement* **else** *statement*
exp → **0** | **1**

String: **if (0) other else other**

Parse Tree:



Syntax Tree:

Left and right derivations

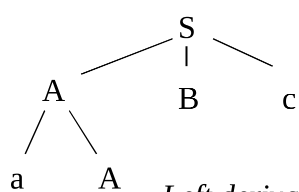
Remember our grammar:

S ::= A B c

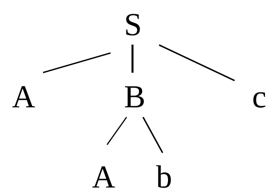
A ::= a A | b

B ::= A b | a

How do we parse the string ***abbbc***?



Left derivation



Right derivation

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 , 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 undesireable.

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

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

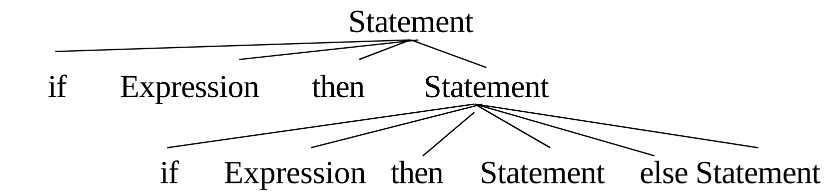
Statement ::= **if** *Expression* **then** *Statement* **else** *Statement*
| **if** *Expression* **then** *Statement*

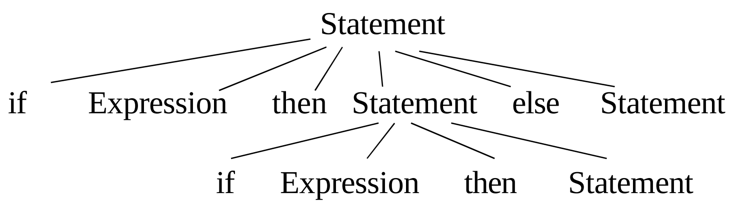
How would you parse the following string?

```
IF x > 0
  THEN IF y > 0
         THEN z := x + y
         ELSE z := x;
```

Is it IF-THEN or IF-THEN-ELSE? (continued)

There are two possible parse trees:

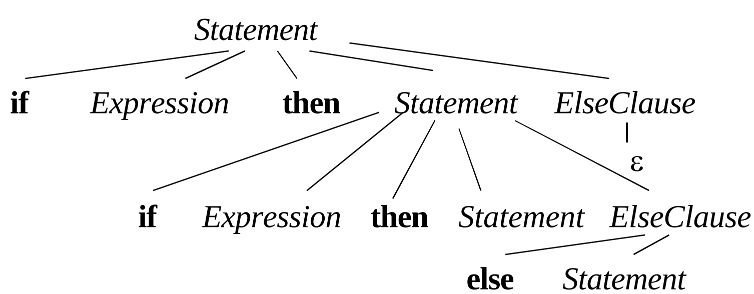




Is it IF-THEN or IF-THEN-ELSE? (continued)

Statement ::= **if** *Expression* **then** *Statement* *ElseClause*

ElseClause ::= **else** *Statement* | ε



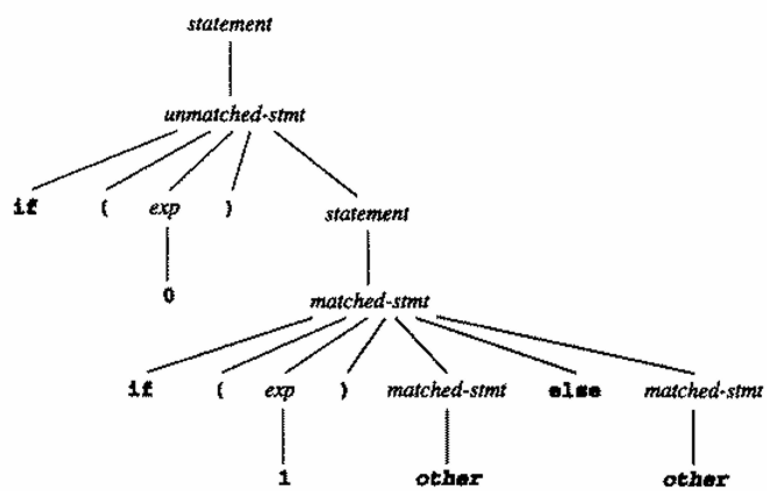
The Dangling Else Problem

Using disambiguating rule called as **most closely nested rule.**

A solution to the dangling else ambiguity in the BNF itself is more difficult than the previous ambiguities we have seen. A solution is as follows:

statement → *matched-stmt* | *unmatched-stmt*
matched-stmt → **if** (*exp*) *matched-stmt* **else** *matched-stmt* | **other**
unmatched-stmt → **if** (*exp*) *statement*
| **if** (*exp*) *matched-stmt* **else** *unmatched-stmt*
exp → **0** | **1**

This works by permitting only a *matched-stmt* to come before an **else** in an if-statement, thus forcing all else-parts to be matched as soon as possible. For instance, the associated parse tree for our sample string now becomes



which indeed associates the else-part with the second if-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**
	*****, /, **DIV**, **MOD**, **AND**
	+, **-**, **OR**
Lowest	=, < >, > =, < =, >, <

Operator Precedence (continued)

This can 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. Add.-op Term | Term

And finally,

Rel.-Expr. ::= Rel.-Expr Rel.-op Expr. | Expr.

Once we add the production

Unary-Factor ::= Identifier | Constant | (Rel.Expr.)

we have a complete expression 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

ExprA ::= ExprA opA ExprB | Exprb

ExprB ::= ExprB opB ExprC | ExprC

....

ExprZ ::= Identifier | Const | ....

Operator Precedence and Associativity

exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | **number**

In the above grammar the operators are set to be left associative and higher precedence operators are set on the lower level.
To do: Add an operator like ^ (power of) in the existing expression grammar which is right associative and has highest precedence among +,- and *.

DEFINING CFG FOR RELATIONAL EXPRESSIONS

A-E → E + T | T

T → T * F | F

F → id | (A-E)

R-E → A-E R-OPR A-E

R-OPR → <|> | <= |>= | == | !=

2 < 8 → valid

a >= h → valid

(a+b) <= (c+d) → valid

CFG for LOGICAL EXPRESSION:

Logical Operators → and, or, not

Binary → and, or

Unary → not

L-E → A-E LOPR-A A-E | L-OPRB A-E

L-OPRA → and| or

L-OPRB → not

(A&&B) || (! B) → valid expression

(a<b) && (a < c) → valid

L-E → E && E | E "|" "|" E | ! E

E → A-E | R-E | L-E

((a>b || a<c) && a<d)

! (a>b && c<d)

! ! ! a   ! ! (a<b)

Expression grammar with compound statement

IF → if (EXPR) STMT ELSE-PART

ELSE-PART → ε | else STMT

EXPR → 0 | 1

STMT → SIMPLE | COMPUND

SIMPLE → ASSIGN-S| IF | FOR | WHILE | Do-WHILE | SWITCH | RETURN | GOTO …

COMPUND → { STMTS }

STMTS → ε | STMT STMTS

CFG of FOR:

FOR → for (EXPR;EXPR ;EXPR) STMT

CFG of WHILE:

WHILE → while (EXPR) STMT

CFG of DO-WHILE:

DO-WHILE → do COMPOUND while (EXPR) ;

DEFINING FUNCTION CALL:

name(parameters)

sum(a,b);

a = sum(a,b)+h;

cout<<sum(a,b)<<endl;

abc(a+B,)

display()

FN-CALL → identifier (PARAM-LIST)

PARAM-LIST → ε | PARAMETERS

PARAMETERS → PARAM | PARAM , PARAMETERS

PARAM → EXPR

DEFINITION of FUNCTION DEFINITION:

2 parts:

Ret-type name(parameters) ➔ header

{

➔ body

}

int sum (int a, int b)

CFG for Function definition:

FN_DEF → FN-HDR FN-BODY

FN-HDR→ RET-TYPE identifier (F-PARAM-LIST)

RET-TYPE → ε | void |int| float| double | char| identifier….

F-PARAM-LIST → ε | F-PARAM MORE-F-PARAM

F-PARAM → PARAM-TYPE identifier

PARAM-TYPE → int | float |double |char |bool…..

MORE-F-PARAM → ε | F-PARAM MORE-F-PARAM

DEFINITION of FUNCTION DECLARATION:

FN-DEC → FN-HDR ;

Define C language SWITCH Statement using BNF Notation. You may assume that EXPR and STMT are already defined.

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 ::= *CondExpr AssnOp AssnExpr* | *CondExpr*

AssnOp ::= **=** | ***=** | **/=** | **%=** | **+=** | **-=** | **<<=** | **>>=** | **&=**
| **^=** | **!=**

CondExpr ::= *LogORExpr* | *LogORExpr* **?** *Expr* **:** *CondExpr*

LogORExpr ::= *LogORExpr* **||** *LogANDExpr* | *LogANDExpr*

LogANDExpr ::= *LogANDExpr* **&&** *InclORExpr* | *InclORExpr*

InclORExpr ::= *InclORExpr* **|** *ExclORExpr* | *ExclORExpr*

Derivation Separator

C operator

Expression grammar in C (continued)

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* ***** Cast*Expr* | *MultExpr* **/** Cast*Expr*
| *MultExpr* **%** Cast*Expr* | *CastExpr*

Expression grammar in C (continued)

CastExpr ::= **(typename)** *CastExpr* | *UnExpr*

UnExpr ::= *PostExpr* | **++***UnExpr* | **--***UnExpr*

| *UnOp CastExpr* | **sizeof** *UnExpr* | **sizeof(typename)**

UnOp ::= **&** | ***** | **+** | **-** | **~** | **!**

ExprList ::= *ExprList***,** *AssnExpr* | *AssnExpr*

PostExpr ::= *PrimExpr* | *PostExpr***[***Expr***]** | *PosrExpr***(***ExprList***)**

| *PostExpr***.id** | Post *Expr* **->** **id** | *PostExpr* **++**

| PostExpr **--**

PrimExpr ::= *Literal* | **(***Expr***)** | **id**

Literal ::= **integer-constant** | **char-constant** | **float-constant**

| **string-constant**