

2 INTRODUCTION: Syntax and Semantics

Semantics Translational and Axiomatic

Syntax Examples

3 Syntax Examples, Grammatical Description Of Syntax

4 BUILDING THE DEFINITION OF A PROGRAMMING LANGUAGE

Defining Programming Language With CFG - letters, identifiers

Defining Programming Language With CFG -numbers

Defining Programming Language With CFG -Precedence Expression Grammar, Procedure Calls

5 Defining Programming Language With CFG Almost Everything

EBNF-Abbreviations of CFG rules

6 SYNTAX CHARTS

7 SUMMARY: Definition Of CFG, of BNF, of EBNF, of Syntax Chart

8 Language Defined By CFG-Derivation

8 Analysis: Relation of Derivation and Parsing

9 Three Representations of Derivations

10 Ambiguity

11 Another Example Of Ambiguity

12 ONE LANGUAGE - TWO DIFFERENT GRAMMARS-Example

13 Parsing Definition, Top-Down Parsing

14 Efficient Top-Down Parsing LL(k) Example

15 Efficient Bottom-Up Left-Right Parsing Rules, LR(K) Example

16 Efficient Bottom-Up Left-Right Parse, LR(K) Example

17 More On Compiler Structure (See Notes I introduction)

18 Optimization Examples

19 The Syntax Analysis Section and Operator Precedence

20 Syntax Analysis: Abstract Machine Language Generation With Precedence Grammar Examples

21 Phrase Structured Grammars; Types 3, 2, 1, and 0

22 Notes On Defining Precedence Grammars

CONTENTS

Syntax and Semantics

Send her to I. Send her to health. Send her to Saturday. Send her to me. Send her to Saturn.

Introduction

First we give informal descriptions of the concepts of syntax and semantics. Later these will be defined more formally. If L is a programming language, then $syn(L)$ represents the syntax of L . $syn(L)$ is the set of all strings of symbols which are legitimate appearing instructions or statements, with correct punctuation, and composition (without regard to any interpretation of the variables except that, in a language which requires declaration, each must be declared), in L . An ordered set of strings, each a legitimate instruction or statement in $syn(L)$, is not required to have these legitimate instructions and statements in an order that makes sense. For example, two identical assignments in sequence in program P does not prevent P from being in $syn(L)$ (for a syntax checker its OK, though an optimizer may replace it with a more efficient one). (ex in English: *Jack and Jill went up the toothpaste.* -Syntax OK, Semantics??)

```
X = 5;
X = 5;
Syntax OK
```

A general way to define semantics is to assign to each set of statements and instructions of a prospective program, S , each in $syn(L)$, an interpretation or meaning, $M(S)$. $M(S)$ may be a sequence of computer actions, or a functional description, or a meaning assigned to the S , The set of pairs $\langle S, M(S) \rangle$ constitute one of the ways of defining the semantics of L . (For many S s the $M(S)$ will simply be "nonsense"). We could define a legitimate programming language to consist of only those strings S , for which $M(S)$ is not "nonsense", but we do not know how to build a Compiler that passes only such "sensible" strings. Also syntactically valid construct can have a number of possible meanings (ex in English. *Sally saw Sam in the park (,) with a telescope.* Did Sam have the telescope or Did Sally use the telescope?)

```
if(x==y)x/y=7;
Syntax NG
```

It is possible to construct $M(S)$ to be a machine language implementation of the higher level language statement S , and to be virtually certain that the resultant computation will be the same i.e., if S , computes a function, $M(S)$, will compute the same function, if S loops forever so will $M(S)$ etc. This can be extended to translating an entire program P in the higher level language to $M(P)$ in machine language. Similarly it may be possible to obtain a Natural Language interpretation of P by translation. (**Translational Semantics**)

If, on the other hand, $M(S)$ is to be of significantly higher level* then we do not know how to do the translation in any general way. If however, we are given S and a proposed higher level $M(S)$ it is sometimes possible to prove their equivalence. This proof process is also referred to as semantics (**Axiomatic Semantics**).



```
program Ex1(input,output);
var X,Y,Z: real
begin
  readln (X,Y);
  Z := 50.0 + Y / (Y + 27.0);
  writeln Z;           write to input
end.
Program in  $syn(\text{Pascal})$ 
(a)
```

```
(define (gcd u v)
  (cond ( (= v 0) u)
        (else gcd v (remainder (u/v) ) )
  )
)
gcd(12, 8) = gcd(8,4) = gcd(4,0) = 4
Program In  $syn(\text{Scheme})$ 
(b)
```

- (a) Removed the **begin** and/or **end**. and this "program" would no longer be in $syn(\text{Pascal})$. It would also not be in $syn(\text{Pascal})$ if the assignment were changed to: $X * Z := 50.0 + X / (Y + 27.0)$; Note that the variable Z is never used-this is inefficient but OK otherwise
- (b) The *Scheme* function definition in (b) is good, its parentheses are balanced and it is otherwise valid-furthermore it does define the gcd (greatest common divisor).

Syntax Examples

```

Here is another program in syn(Pascal)
program
Ex1(input,output);
var Y,Z: real;
begin
  writeln (Z);
  Z := 50.0 + Y - (Y + 50.0);
  readln ( Z );
end.
Program in syn(Pascal)
(a)

```

```

Not in syn(Scheme) and even if it were it is semantically
bad
1(define 2(gcd u v)
  2(cond 3((v = 0) u)
    3 4 else 3 4 gcd 5 remainder(u/v) v 4 ← parenthesis
  ) 3
2)
Program not in syn(Scheme)
(b)

```

(a) Is syntactically and semantically correct but flawed inefficient. Z =0 could replace the assignment
 (b) has unbalanced parenthesis. Even if these were balanced it also is semantically flawed because but evaluation yields gcd(12, 8) = gcd(4,8) = gcd(4,8), etc. it is semantically silly, i.e., *M(I)* i.e., “nonsense”. Notice that we check the semantics by “running it”-which is a legitimate approach.

Syntax Examples

Grammatical Description Of Syntax

Formal Syntax: Levels of Description: A generative grammar provides a formal system for defining sets of strings and thus of defining the syntax of a language. Before describing such a grammar we give some explicit examples of the specification of syntax using Context Free Grammars (CFG), or equivalently Backus Naur Form (BNF). (Introduced by N. Chomsky in a study of languages and quite succesfull for Programming Languages, less so for Natural Languages.) (used first in defining ALGOL60), and/or some extensions of these, Extended BNF (EBNF). These extensions are essentially abbreviations—they do not allow the definition of languages syntax not achievable with CFG or BNF, but do allow the definitions to be more concise. For the purpose of this course all the symbolism allowed in EBNF, should be understood-not what is in each of the subsets CFG, and BNF and These abbreviation will be introduce in the following examples.

CFG/BNF: Starting with the grammatical definition of simple syntactic constituents of programming languages the more complex components of the language are developed. Each level is based on those developed earlier.

Basic Lower Level Lexical Units are (“letters” and “digits”)

- 0,1,2,3,4,... are called <digit>
- A,B,C,D,E,... are called <bigletter>
- a,b,c,d,e,...,.,/,[,],.,!,@,#,+,(),... are called <smallletter+> [other characters]

Higher level Construct (Built From The Lower Level Units) (words, string)

- begin, end, else, then,... are <keywords>
- x := sam, ... are <statements>

The CFG grammar for <digit>, <bigletter> and <smallletter+> is sketched below

<digit> ----> 0	<bigletter> ----> A
<digit> ----> 1	<bigletter> ----> B
<digit> ----> 2	.
<digit> ----> 3	.
.	
<smallletter+>----> a	
<smallletter+>----> b	
.	
<smallletter+>----> +	
<smallletter+>----> #	

<X> ----> y
 <X>: <X > contains the letter “y”
Reading CFG

Defining Programming Language With CFG

Assuming we have already completely defined <bigletters>, and <smallletters>, the set of all letters, designated <letters> is defined in a CFG by:

<letter> ----> <bigletter>
 <letter> ----> <smallletter+>
 or more compactly:

<X> ----> <Y>a<Z>
 <X>: <X> contains every string made from concatenating a member of <Y> with "a" followed by a member of <Z>
 <X> ----> <Y>a<Z> | +<Q>
 <X> contains all strings above also "+" followed by a member of <Q>.
 Abbreviation | is found in BNF and EBNF not CFG

Reading CFG/BNF

<letters> ----> <bigletter> | <smallletter+>

Defining Programming Language With CFG+Abbreviations - letters

Set of all symbolic addresses and variables is <id>

<id> ----><letter>
 <id> ----> <id> <letter>
 <id> ----> <id> <digit>

an <id> = 1 letter or
 an <id>(1 letter) followed by 1 letter=2 letters or
 an <id>(2 letters) followed by 1 letter=3 letters or
 an <id>(k letters) followed by 1 letter = k+1 letters
 or an<id>(k letters) followed by a digit

Reading CFG (Recursion)

<id> ----> <letter> { <letter> | <digits> } [Defining the same strings using the extensions |, and the EBNF construct { x } meaning repeatedly concatenate x, 0, or 1, or 2, . . . times:

Defining Programming Language With CFG+Abbreviations - Identifiers

Set of counting numbers is <integer>

< digit> ----> 0|1|2|...
 <usinteger> ----><digit> | <integer><digit>
 <sinteger> ----> [+ | -] <integer>
 <integer> ----> <usinteger> |

The EBNF extensions [x], means: repeat x either 0 or 1 times, [x | y] any string of xs and ys or ε=null string

Reading BNF/EBNF

Set of computing numbers <decimals>

<decimal> ----> <integer>
 <decimal> ----> <integer> . <integer>

Set of all computing numbers <reals>

<real> ----> <integer> E <integer>
 <real> ----> <decimal> E <sinteger>
 <numb> ----> <integer> | <decimal> | <real>

Defining Programming Language With CFG+ Abbreviations - Numbers

Highest Levels ("sentences, paragraphs, ")

<exp> ----> <fac> | <exp> + <fac> | <exp> - <fac> [<exp>. = one, or a sum/subtracts of <fac>s]
 <fac> ----> <op> | <fac> * <op> | <fac> / <op> [<fac> = one, or a product/division of operands and parenthesized expression]
 <op> ----> <numb> | <id> | (<exp>)

Procedure Calls

<act-param> ----><exp>| <proc-stat>
 <proc-id> ----> <id>
 <proc-stat> ----> <proc-id> | <proc-id> (<param-list>)
 <par-list> ----> <act-param> | <act-par> , <param-list>

Defining Programming Language With CFG+Abbreviations Precedence Expression Grammar

Building The Definition Of A Programming Language

“The chapter, the book”

<program> ---> <declarations> <stmtnt>
<stmtnt> ---> <nm> := <exp> | <nm> (<exp-1st>)

<stmtnt> ---> **begin** <stmtnt-1st> **end**
<stmtnt> ---> **if** <b-exp> **then** <stmtnt>;
<stmtnt> ---> **if** <b-exp> **then** <stmtnt> **else** <stmtnt>;
<stmtnt> ---> **case** <exp> **of** <cases> **end**;
<stmtnt> ---> **while** <b-exp> **do** <stmtnt>
<stmtnt> ---> **repeat** <stmtnt-1st> **until** <b-exp>
<stmtnt> ---> **for** <nm> := <exp> **to** <exp> **do** <stmtnt>

<stmtnt-1st> ---> <empty> | <stmtnt> ; <stmtnt-1st>
<cases> ---> <const> ; <stmtnt>

<proc-stat> ---> <proc-id> | <proc-id> (<param-list>)
<param-list> ---> <act-param> | <act-par> , <param-list>
<act-param> ---> <exp> | <proc-stat>
<proc-id> ---> <id>

<exp> ---> <fac> | <exp> + <fac> | <exp> - <fac>
<fac> ---> <operand> | <fac> * <operand> | <fac> | <operand>
<operand> ---> <numb> | <id> j (<exp>)

integer> ---> <digit> | <integer><digit>
<decimal> ---> <integer> | <integer> . <integer>
<real> ---> <integer> E <integer> | <decimal> E <sinteger>
<numb> ---> <integer> | <decimal> | <real>
<numb> ---> 0 | 1 | 2 | 3 ... | 9

<b-exp> ---> <b-fac> | <b-exp> || <b-fac>
<b-fac> ---> <b-op> | <b-exp> && <b-fac>
<b-op> ---> <conditional> | <id> (<b-exp>)

<conditional> ---> <exp> (== | != | > | >=) <exp>

DEFINING PROGRAMMING LANGUAGE-With Previous Definitions = Almost Everything

EBNF-ABBREVIATIONS

: (α) indicates grouping, of the enclosed string, α , It is to be distinguished from the use of parenthesis to represent the terminal symbols “(” and “)”. The latter use is indicated by use of bold face and italics. The construct can be used to shorten a right-side. For example the right-side “<a>x | x” can be written as (<a> |) x .

{ α } indicates 0 or any number of occurrence of the enclosed string, α , those enclose

<op> → <numb> | <id> | (<exp>)
<fac> → <op> | <fac> (* | /) <op>
<exp> → <fac> { (+ | -) <fac> }

<act-param> → <exp> | <proc-stat>

<proc-id> → <id>

<proc-stat> → <proc-id> | <proc-id> (<act-param> { , <act-param> })

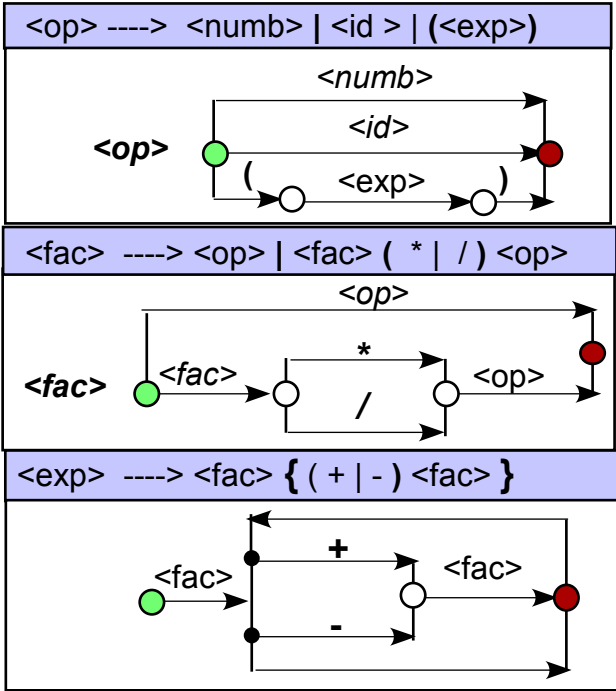
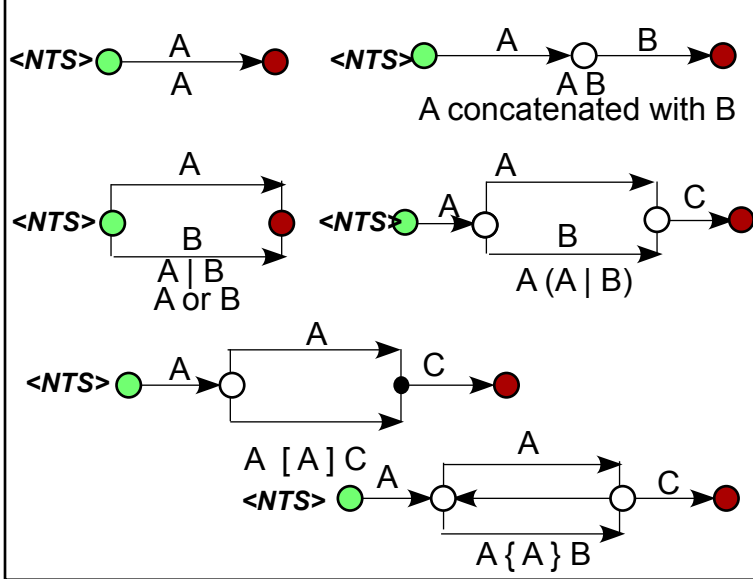
- An <exp> is <fac> followed by nothing or by either a terminal symbol “+” or “-” followed by a <fac>

Reading EBNF (Recursion)

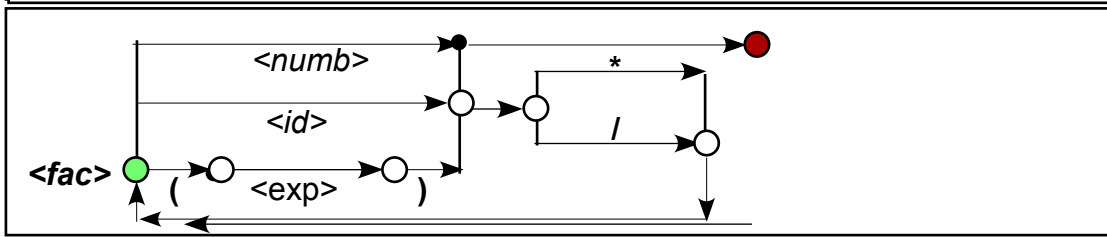
SYNTAX CHARTS- Pictorial Representation Of EBNF

A Syntax Chart is a pictorial representation of EBNF. In the figure (a) Syntax Charts for basic EBNF constructs are given. In (b) Syntax Charts for the right-sides of the three rule above are given.

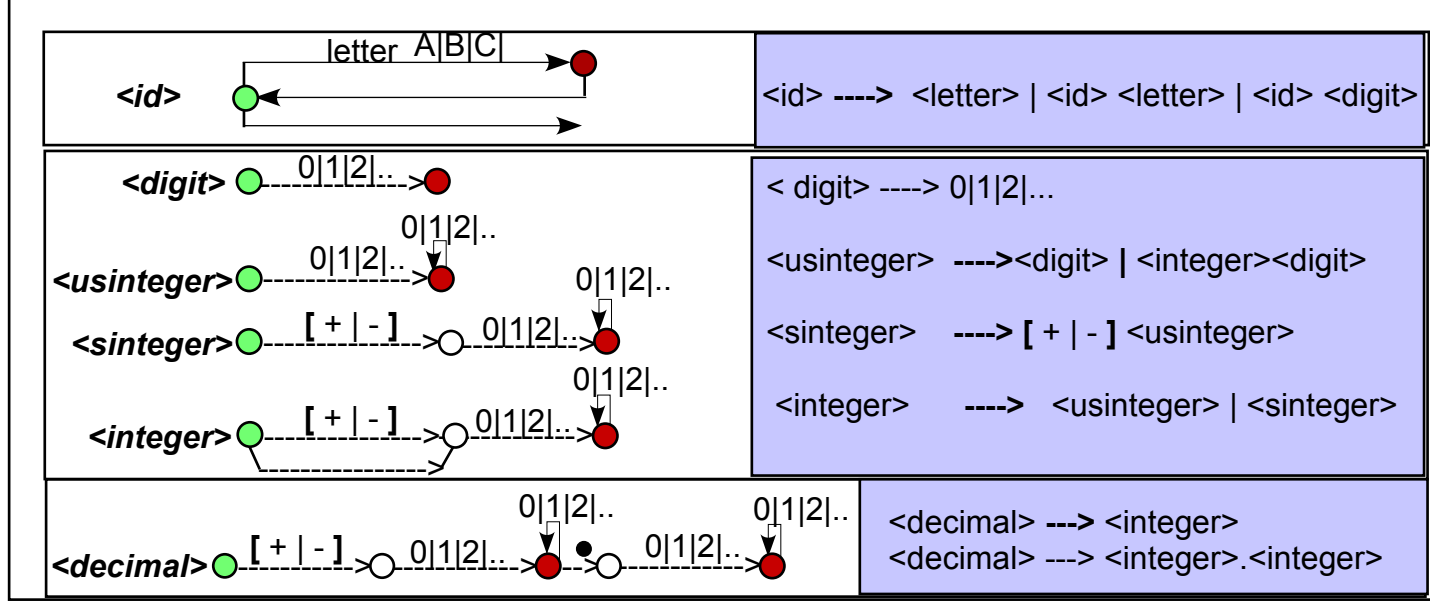
If A, B and C are: 1, a Non-Terminal symbols, or 2, Terminal symbols or 3, an Inserted **Syntax Charts**, then Any of the following are **Syntax Charts**



$\langle \text{fac} \rangle \rightarrow \langle \text{op} \rangle \mid \langle \text{fac} \rangle (* \mid /) \langle \text{op} \rangle =$
 $\langle \text{fac} \rangle \rightarrow \langle \text{op} \rangle \mid \langle \text{fac} \rangle (* \mid /) \langle \text{op} \rangle (* \mid /) \langle \text{op} \rangle$ By Substitution
 $\langle \text{fac} \rangle \rightarrow \langle \text{op} \rangle \mid \langle \text{fac} \rangle (* \mid /) \dots \langle \text{op} \rangle (* \mid /) \langle \text{op} \rangle$ By Substitution
 $\langle \text{fac} \rangle \rightarrow \langle \text{op} \rangle \{ (* \mid /) \langle \text{op} \rangle \} =$ EBNF
 $\langle \text{fac} \rangle \rightarrow \langle \text{num} \rangle \mid \langle \text{id} \rangle \mid (\langle \text{exp} \rangle) \mid \{ (* \mid /) (\langle \text{num} \rangle \mid \langle \text{id} \rangle \mid (\langle \text{exp} \rangle)) \}$



In graph each path from green circle to red circle represents a string in L. So if edges are all in terminal symbols a string, S, would be in L if there was a path from green circle to red circle



SYNTAX CHARTS and FSMs

SUMMARY:

Definition Of **CFG**, of **BNF**, of **EBNF**, of **Syntax Chart**

A **CFG** consists of:

1. A set of **non-terminal** symbols (**NTS's**)
2. A finite set of **terminal** symbols (**TS's**)
3. A finite set of production rules written $x \rightarrow R$ where x is NTS, R (right-side) is a string of TS's and NTS's
4. A distinguished NTS called the **starting symbol** (also goal or distinguished symbol)
[The symbol to the left of an arrow is an NTS, All NTSs that appear on the right of a rule also appear on the left at least once in a reasonable grammar]

Notation

Commonly an NTS is represented by a bracketed name i.e., " $\langle name \rangle$ ", and an TS by a literal copy of itself, for key TS's in the grammar such as $\langle, \rangle, [,], (,)$ etc. a special indication, perhaps an underline, or boldface, is needed to distinguish it from the special operations introduced by EBNF. In another common notation NTS's are given by capitalized letters and TS's by using symbols to represent themselves except capitals, or TS's and strings of TS's are quoted. Again it is necessary to make exceptions when grammar notation, i.e., capitals for NTS's, or quotes, and literal representation clash.

A **BNF** Grammar is made up of:

1. CFG
2. And the use of $|$ ("or") to indicate alternative right-sides. So $x \rightarrow R_1 | R_2$ represents $x \rightarrow R_1$ and $x \rightarrow R_2$

An **EBNF** Grammar is made up of:

1. BNF
2. $()$'s for grouping "or's" so as to distribute concatenation through them.
3. $\{ \alpha \}$ represents 0 or more concatenations of α
4. $[\alpha]$ represents 0 or 1 concatenations of α
5. Normally concatenation takes precedence over $|$. Parenthesis can be used to force an $|$ to take precedence over a related concatenation. Parentheses are used for grouping "ors" so as to distribute concatenation through the them.

A **Syntax Chart** is a direct translation of EBNF to graphical notation, See figure 1.

1. It is a series-parallel graph in which directed edges connect adjacent concatenated TS and NTS's.
2. The "or" operation in: $X | Y$ is translated to two parallel paths one for X and one for Y .
3. The square brackets in $[X]$ is represented with the syntax chart representation of X with an added directed edges from before the representation of X to just after it
4. The curly brackets in $\{ X \}$ is represented with the syntax chart representation of X with two added directed edges: One from before the representation of X to just after it and another from just after the representation of X back to just before.
5. The syntax chart for the strings included within all unested parenthesis are constructed and then these are considered unparenthesized units in the construction of the total syntax chart. This process is repeated until there are no parentheses remaining.

Language Defined By CFG-Derivation

A CFG, G , consists a set of production rules. How then does it define a language $L(G)$? It defines a language through the notion of a **leftmost derivation**.

Left-Most Derivation : Basic

A **leftmost derivation** in grammar G . is a sequence of **sentential forms**. A sentential form is a string of intermixed TSs and NTSs. The first sentential form is the starting symbol, say Z , The next is produced by substituting for Z the right side of a rule with Z on the left. In general then one finds the **leftmost NTS in the latest sentential form, S_n** , say A and replaces it with the right side of a rule with A on the left, say $A \rightarrow \alpha$, to get the next sentential form, S_{n+1} , There is no substitution for TS's,. The derivation terminates with a sentential form consisting of TSs only (a terminal string). With Z being the starting symbol and τ being a terminal string, and each **intermediate sentential** form having at least one NTS, then the leftmost derivation can be represented by giving the result of each rule used and the sentential forms that result as follows:

$$\alpha_0 = Z \xrightarrow{r_1} \alpha_1 \xrightarrow{r_2} \alpha_2 \xrightarrow{r_3} \dots \xrightarrow{r_{n-1}} \alpha_{n-1} \xrightarrow{r_n} \alpha_n \rightarrow \tau$$

Four Representatives of the Left-Most Derivation

- The derivation is completely **defined by the sequence of sentential forms** it generates in arriving at τ starting with Z , i.e. $\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_n, \tau$.
- Alternatively the derivation is completely **defined by the sequence of rules** used in arriving at τ starting with Z , i.e., $r_1, r_2, r_3, \dots, r_{n-1}$
- A third way involves **an oriented tree** with every node labelled with a symbol, either an NTS or TS. If a node is TS then it is a terminal node of the tree. If a node, x , is labelled with an NTS, say A , its childrens are determined from a rule of the CFG with A on the left, of the form $A \rightarrow \alpha$. There is a child node for each symbol in α , labelled by that symbol, and arranged left to right in the order they appear in α .

Using the rules given previously for *<decimal>*, *<integer>*, and *<digit>*, and repeated in the next figure, an example of three different ways to represent a derivation are given. The derivation(s) in the figure are all for the decimal number 19.2 with *<decimal>* as the starting symbol.

- The **Sentential form** of the leftmost derivation, then
 - the **Rule sequence** of that derivation (at the left of each step in the previous derivation), and finally
 - the **Tree** (which is not particularly "left-most" , but represents the same derivations in that the same substitution are made for the same NTSs occurrences as in the "leftmost derivation")
- [d) The **Parenthesis Form- like the tree**]

The Language Defined By a CFG

The language defined by grammar G , $L(G)$. is the set of all strings that can be generated by a leftmost derivation from that grammar. So for the decimal number example all strings of terminals that can be derived from *<decomal>* by any of the three techniques constitute $L(G)$. Referring to all the different ways considered above for language definition:

- Every string whose derivation can be given in one of the forms above can also be given by both of the others.
- The syntax language defined by grammar G , $L(G)$., is the set of all terminal strings that can be derived in G .

Analysis: Relation of Derivation and Parsing

Given a grammar G describing the syntax of a language, $L(G)$, enables a compiler to read a string of symbols, τ supposedly in $L(G)$ and to determine the derivation by which τ was derived in G , that is, to determine the sequence of rules, or the sequence of sentential forms, or the parse tree by which τ could have been derived. (The derivation should be unique-only one rule sequence could derivet .)

THE GRAMMAR

rule #\	1	$\langle decimal \rangle \rightarrow \langle integer \rangle . \langle integer \rangle .$
	2	$\langle integer \rangle \rightarrow \langle integer \rangle \langle digit \rangle$
	3	$\langle integer \rangle \rightarrow \langle digit \rangle$
	4	$\langle digits \rangle \rightarrow 0$
	5	$\langle digits \rangle \rightarrow 1$
	6	$\langle digits \rangle \rightarrow 2$
	13	$\langle digits \rangle \rightarrow 9$

A CFG, G $\langle decimal \rangle$, and its Syntax Chart Form

THE LEFTMOST DERIVATION

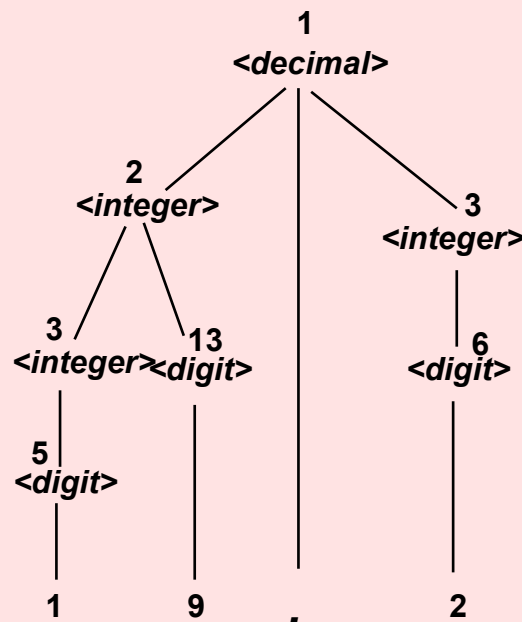
a) $\langle decimal \rangle \rightarrow \langle integer \rangle . \langle integer \rangle \rightarrow \langle integer \rangle . \langle digit \rangle . \langle integer \rangle \rightarrow \langle digit \rangle . \langle digit \rangle . \langle integer \rangle$

b)	1	2	3	
	$\rightarrow 1$	$\langle digit \rangle$	$\rightarrow 19$	$\langle integer \rangle$
		$\rightarrow 19$	$\langle digit \rangle$	$\rightarrow 19.2$
	5	13	3	6

a) Sentential Forms

b) Rule #s

c)



c) Tree Form

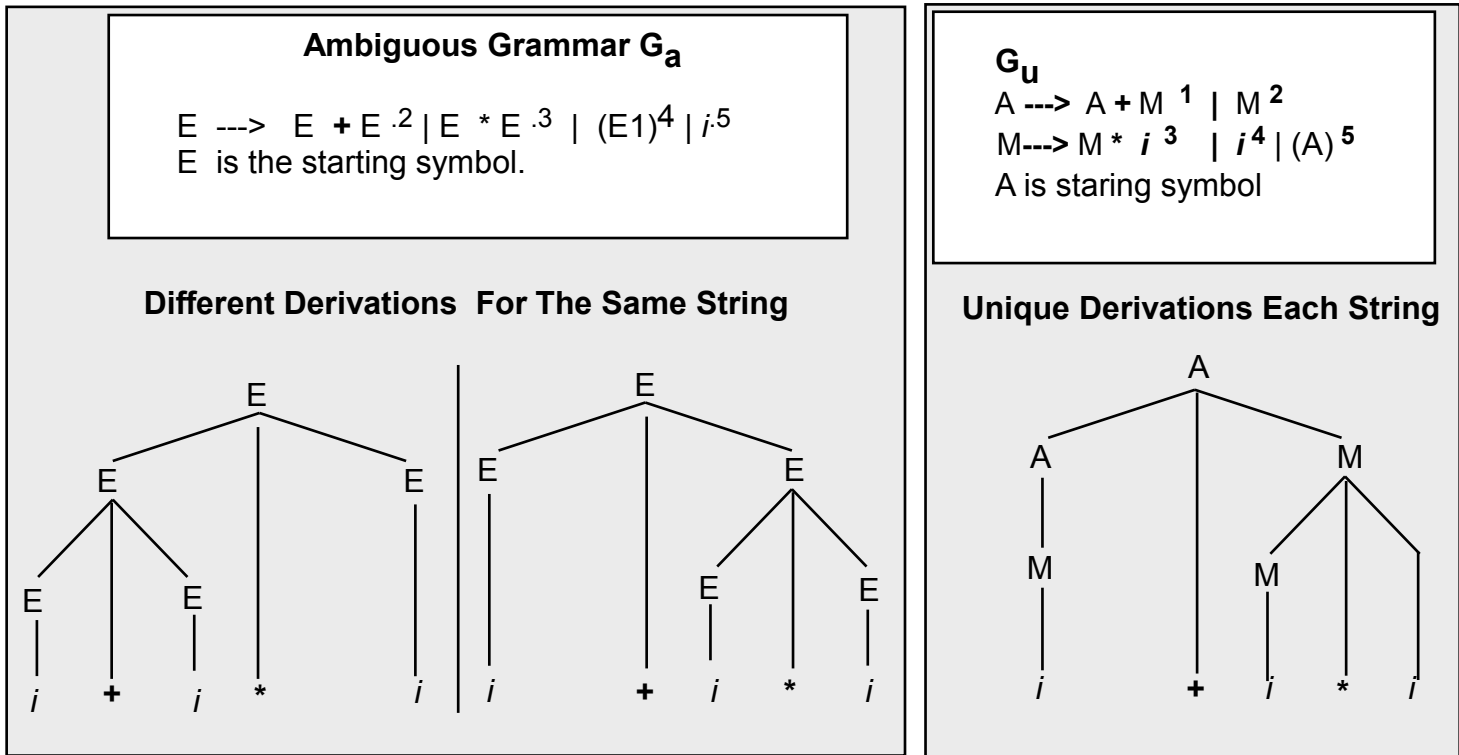
d) $1 [2 [[1] [9]] . 3 [[2]]]$ d) Parenthesis Form

LEFT-MOST DERIVATION IN: a) Sentential Form and in
b) Rule Sequence Form
c) Tree Form
d) Parenthesis Form

EXAMPLE OF DIFFERENT DERIVATION REPRESENTATIONS

Ambiguity

A grammar G , defines a language (all the strings that can be derived from G). The derivation = the parse of a string, is useful in translating to machine language. only if the parse of each string S is unique, but for some grammars there are multiple parses for the same string. Such grammars are ambiguous. They will not serve the purpose of the Compiler. An ambiguous grammar G_a (capital letters are NTSs small letters are TSs) is given along with two derivations (parse) of the same string $S = "i + i * i"$. Also an equivalent unambiguous grammar G_u is shown with its derivation (parse) of S .



G_u is a grammar equivalent to G_a in that it defines the same language as G_a , without ambiguity.

Note some of the properties of G_u which will help in the development of a Parser to be described later for the language of G_u

In G_u an addition, $+$, can only be parsed bottom-up as an A and can only have an addition to its immediate left that was parsed before it (because the $A + M$ right side of an A rule is the only one with a $+$ in it.) A can have a multiplication, a simple i , or a parenthesized expression, (A) , parsed bottom up before it to the right of the $+$. Furthermore the derivation of a sequence of additions i.e.,

$$A \rightarrow A + M \rightarrow A + M + M \rightarrow A + M + M + M \dots$$

always produces the $+$ s from right to left and therefore in a bottom-up parse always reduces the rightmost $+$ s first. As we will see the compiler will typically interpret such a parse to mean that the rightmost additions in a string of additions should be done first (right associativity).

A similar interpretation will be given to the bottom-up parse of a sequence of $*$ s. each sequence of multiplications $*$ s must be generated from an M . If the grammar rule for $+$ had been $A \rightarrow M + A \mid M$ instead, the same language would result but a sequence of additions would be interpreted as being left associative. Notice that in the ambiguous grammar, G_a right and left association are both possible in a bottom up parse. Although for $+$ this would not make any difference, if there were a division sign ($/$) in place of the $+$, the choice of associativity effects the result.

Another Example Of Ambiguity

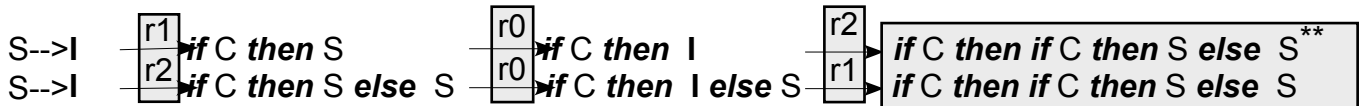
A simple attempt to introduce an *if-then*, and *if-then-else* statement in a CFG, via the NTS I, might use the following grammar fragment. S is any statement, C is any condition, and I is given next. This is in fact the Pascal definition

```
G1: S → I [r0]
     S → <other expressions>
     I → if C then S [r1]
     I → if C then S else S [r2]
```

This produces an ambiguity which is shown by the two derivations below:

There are clearly two different derivations of the string *if C then if C then S else S*.

These are source of the ambiguity:



So the Pascal definition is ambiguous.* The ambiguity yields different semantics for this statement.

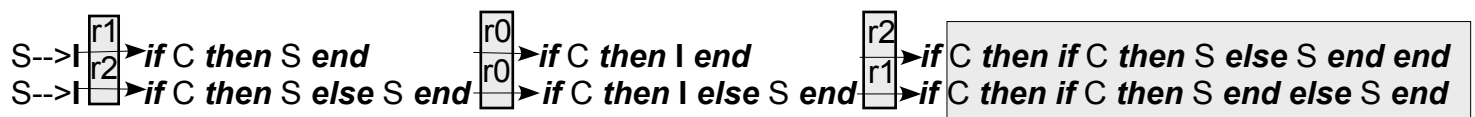
The two derivations corresponds to the two interpretations of the correct order of execution. This is represented by the bracketing below. The square brackets are not part of the language they indicate the order of execution.

1. *if C then [if C then S] else S*
2. *if C then [if C then S else S]*

The simplest fix is that used in the language Modula-2.

```
G2: S → I [r0]
     S → <other expressions>
     I → if C then S end [r1]
     I → if C then S else S end [r2]
```

Now the derivations using the corresponding rules to those above yield two different strings, with two different interpretations as indicated below

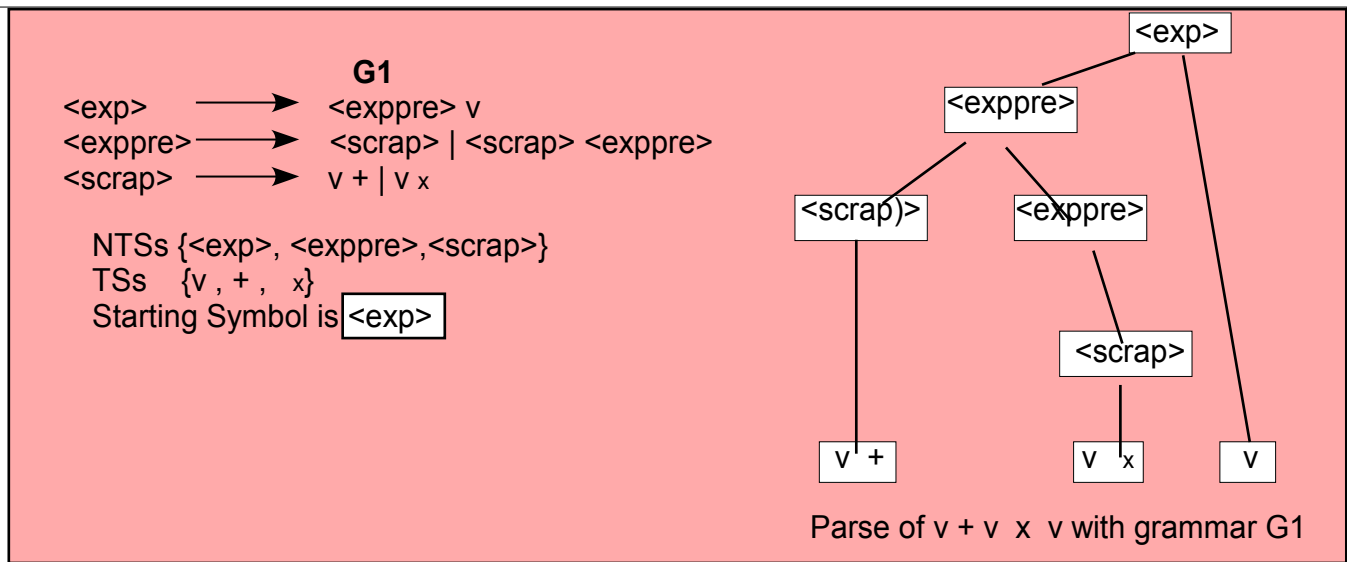


If we interpret the *if.....end* as brackets the interpretations for the two derivations are different- there is no ambiguity.

1. [*if C then [if C then S else S end] end*]
2. [*if C then [if C then S end] else S end*]

* A grammar which only gives 1 parse for the output at ** can be obtained, but then the user would have to know which of the two possible interpretation was valid. Expressing these interpretation in a way that is easily identified requires the effective bracketing achieved by the **end** key word

For a given Programming Language Ambiguous Grammars should be avoided. Furthermore there will generally be many unambiguous grammars for a given Programming Language. From these one must select one which give the most meaningful parse-one translatable to machine language and also can be parsed efficiently G1 and G2 below generate the same language- but for translating purposes G2 is useful while G1 is not



The Parse Can Be Done Easily Top Down But it is not useful

Is $v + v x v$ is in $\langle \text{exp} \rangle$?

it is iff $v + v x$ is in $\langle \text{exppre} \rangle$ [since $\langle \text{exp} \rangle \rightarrow \langle \text{exppre} \rangle v$]

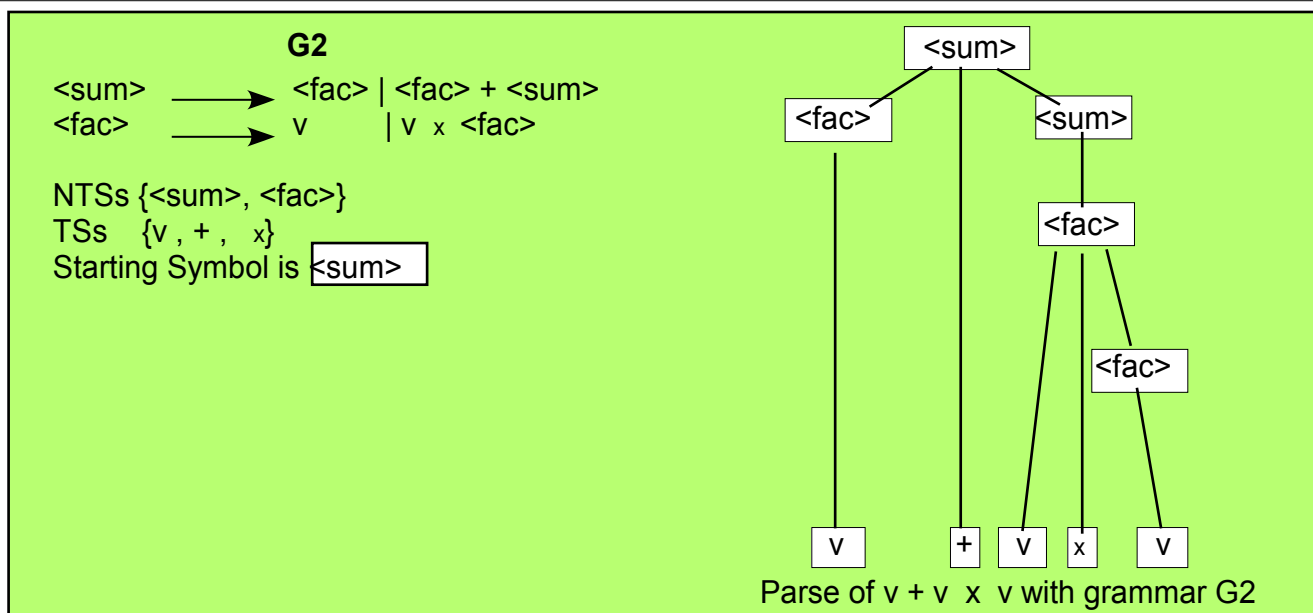
ok if $v + v x$ is in $\langle \text{scrap} \rangle$ [since $\langle \text{exppre} \rangle \rightarrow \langle \text{scrap} \rangle$] *Fails* or

ok if $v + v x$ is in $\langle \text{scrap} \rangle \langle \text{exppre} \rangle$ [since $\langle \text{exppre} \rangle \rightarrow \langle \text{scrap} \rangle \langle \text{exppre} \rangle$]

ok if $v x$ is a member of $\langle \text{exppre} \rangle$ [since $\langle \text{scrap} \rangle \rightarrow v +$]

ok if $v x$ is a member of $\langle \text{scrap} \rangle$ [since $\langle \text{exppre} \rangle \rightarrow \langle \text{scrap} \rangle$]

Ok IT IS
since $\langle \text{scrap} \rangle \rightarrow v +$]



$LG1 = LG2 =$ any infix arithmetic expression with operators $+$ and $*$ and the variable v , ex. $v + v * v * v + v$. However the Parse of G1 is not useful for generating Machine Code while that of G2 is.

ONE LANGUAGE - TWO DIFFERENT GRAMMARS-Example

Parsing Definition

$\underline{\text{parse}}(G, \omega) = 0$ if ω is not deriveable from G , i.e., ω is not in $L(G)$.
= the derivation of ω from G (as a rule sequence or parse tree or etc.) if ω is deriveable from G . (Sometimes there is more than 1 parse- (Ambiguity))

For some purposes we may also be interested in the simpler **recognition**(G, ω) in which:

$\underline{\text{recognition}}(G, \omega) = 0$ if ω is not deriveable from G , i.e., ω is not in $L(G)$.
= 1 if ω is deriveable from G , i.e., ω is in $L(G)$.

Parsing Algorithms

In the worst case parsing an input string for an unambiguous CFG requires $O(n^2)$ time (Early's Algorithm). However parsing with the Grammars actually used to describe most Programming Languages the complexity is $O(n)$. These $O(n)$ Parsers include range from simple Finite State Machines to Push-Down Automata. We give examples from two general classes, Top Down and Bottom Up Parsers..

Top-Down Parsing

There are a variety of algorithms for parsing. In **topdown** parsing one starts with the distinguished (starting) symbol and tries applying rules which may be the first in a derivation of ω the string to be parsed.. Then for the second step it can try all possible second derivation steps, etc. (T

It is important to have some way of limiting the rules to be tried at each step. This is done by continually comparing the sentential form, particularly the terminal symbols therein that will be produced as a result of using a rule with ω , In this comparison corresponding terminal symbols must match or the proposed derivation step is not viable.

Efficient Top-Down Parsing LL(k)

For a certain subclass of CFGs the top-down parse can be done in linear time $O(n)$. Consider the right side of a rule, say $R : X \rightarrow r_1 \dots$, where the first symbol on the right, r_1 , is either a TS or a NTS. Derivations that start with rule R must generate terminal strings that start with r_1 if r_1 is a TS, or deriveable from r_1 if it is an NTS. It is often easy to find all the terminal strings whose first symbol is deriveable from r_1 , say these are $First(r_1)$. (Certainly if r_1 is a terminal symbol $First(r_1) = r_1$). In a top down parse we need only choose rules whose right sides have a first symbol x such that $First(x)$ matches the next symbol in ω to be matched. If this test is sufficient to always select a single rule to be used next in the top-down parse then the grammar is said to be $LL(1)$ (Left Lookahead 1). There is a natural generalization to the definition $LL(k)$ grammar.

The following is an example of a $LL(3)$ grammar (note this is effectively a Regular Grammar G):

GRAMMAR G

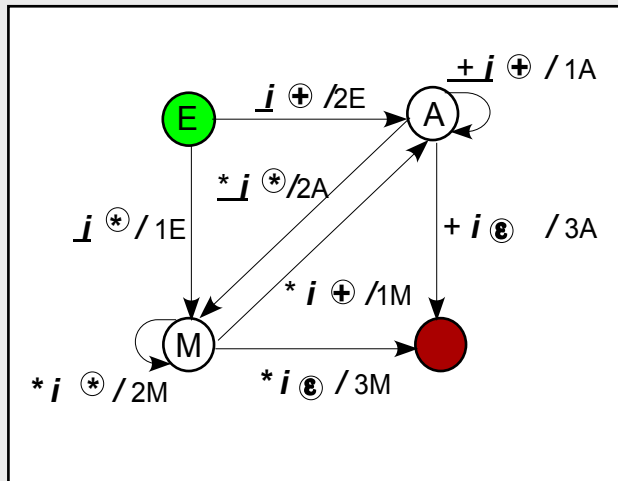
$E \rightarrow i M$ (rule 1E) | $i A$ (rule 2E)
 $A \rightarrow + i A$ (rule 1A) | $+ i M$ (rule 2A) | $+ i$ (rule 3A)
 $M \rightarrow * i A$ (rule 1M) | $* i M$ (rule 2M) | $* i$ (rule 3M)
 E is the starting symbol.

LL(3) PARSING RULES FOR G

For E
 if $i +$ use rule 2E, - i
 if $i *$ use rule 1E, - i

For A
 if $+ i +$ use rule 1A, - $+ i$
 if $+ i *$ use rule 2A, - $+ i$
 if $+ i$ use rule 3A, - $+ i$

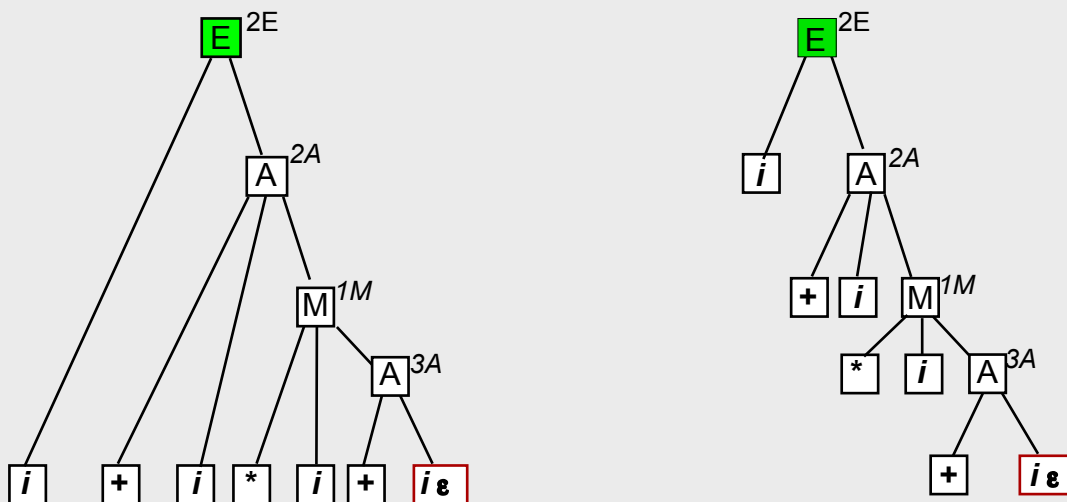
For M
 if $* i +$ use rule 1M - $* i$
 if $* i *$ use rule 2M - $* i$
 if $* i$ use rule 3M, - $* i$



E has 2 children. A and M each have 3 children

if E is the parent and the first 2 inputs are $i+$ then i and A (left to right) are the children of E by rule 2E. For the children of A we look at the 2nd, 3rd and 4th inputs

if A is the parent and we are considering the next 3 consecutive the k th, $k+1$ st and $k+2$ nd inputs and they are $+i+$ then A has three children: $+$ then i then A . For the children of A we look at the $k+2$ nd, $k+3$ rd and 4th inputs



Efficient Top-Down Parsing LL(k) Example

Efficient Bottom-Left-Right Parsing, LR(K)

It is also often possible to build a parser which works **bottom-up and left to right**. The grammar in the next figure is one for which this is possible. Notice that G and G' generate the same language. This grammar is of the type called **LR[k]**. It can be parsed bottom-up and left to right in $O(n)$ time. It reads inputs symbols into a stack. If it decides that the symbols at the top of the stack form the right side of a rule (by looking ahead at the input to be parsed). It replaces those members in the stack with the left side of that rule, and continues. In making the decision it looks up to k characters ahead in the input string. In fact the grammar above is LR[3] and the details of the algorithm are outlined below

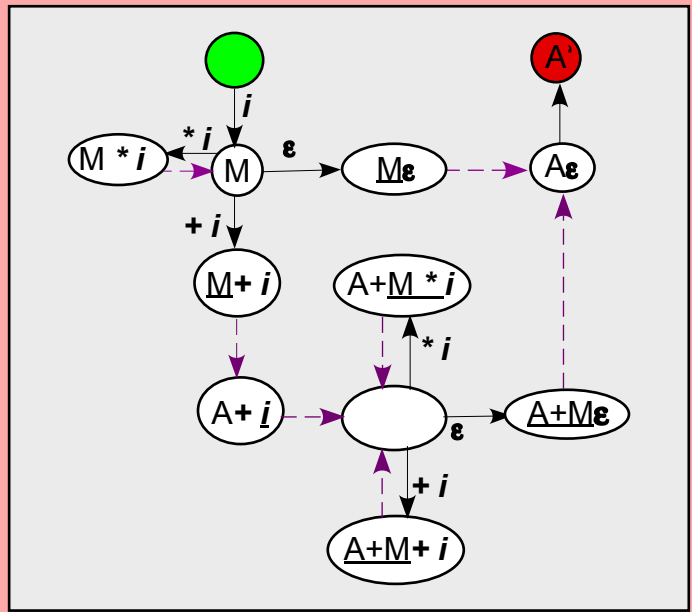
GRAMMAR
 G_U
 $A' \rightarrow A \epsilon$
 $A \rightarrow A + M^1 \mid M^2$
 $M \rightarrow M * i^3 \mid i^4$
 A is the starting symbol.

LR(3) PARSING RULES FOR G_U

- RULES**
- : Leading i is replaced with M
 - M1: If M is followed by $+ i$
 reduce M to A
 - M2: If M is followed by $* i$
 reduce $M * i$ to M
 (push in $* i$)
 - M3: If M is followed by ϵ ,
 reduce M to A

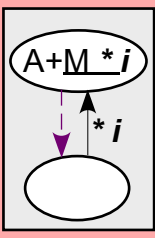
 - A1 If A is followed by $+ i$
 reduce $A + i$ to $A + M$
 (push in $+ i$)
 - A2 If A cannot be followed by $* i$
 - A3 If A is followed by ϵ
 reduce $A \epsilon$ to A'
 (push in ϵ)

 - A+M1 If $A+M$ is followed by $* i$
 reduce $A + \underline{M} * i$ to $A + M$
 (push in $* i$)
 - A+M2 If $A+M$ is followed by $+ i$
 reduce $\underline{A} + M + i$ to $A + i$
 - A+M3 If $A+M$ is followed by ϵ
 reduce $A + M \epsilon$ to $A \epsilon$

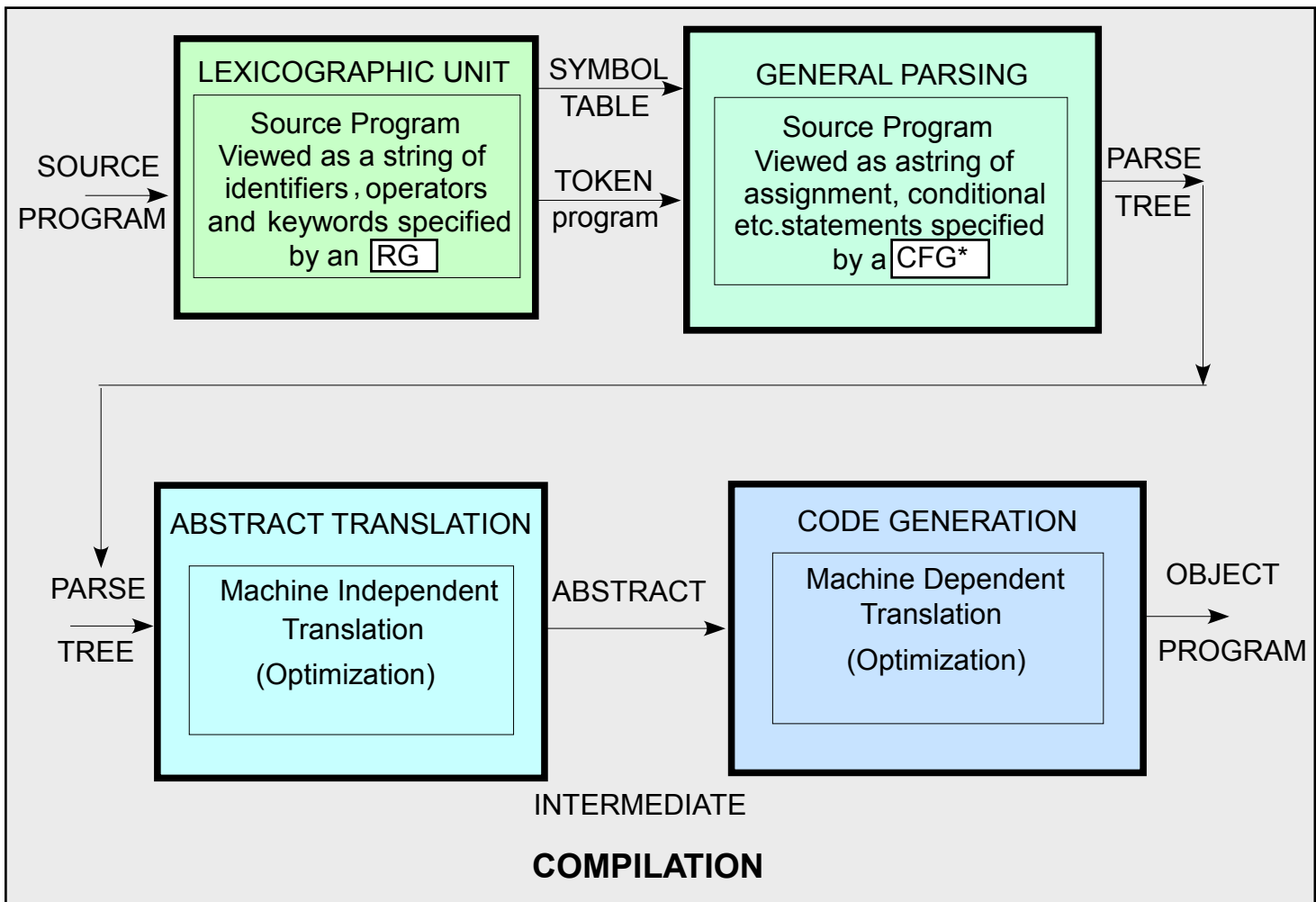


The set of symbols currently in the stack is in each ellipse.

There are 2 types of transition shown.
 1) On each labelled black arrow the next inputs detected in the input string sufficient for making a decision are given.
 2) Each unlabelled dashed arrow goes from a set of children symbols to a parent of some of those children (those underlined).



Efficient Bottom-Up Left-Right Parsing Rules, LR(K) Example



More On Compiler Structure (See Notes I introduction)

A little deeper analysis of compilation than given earlier will reveal the role of different grammar types. As said in the previous chapter, Typically compiler design is broken into four parts, the Lexicographic, General Parsing, Abstract Translation and Code Generation Units, as seen above.

Lexicographic Unit

To define the language perceived by the **Lexicographic unit** a very simple CFG called an RG or RE is adequate, and a simple machine, to be described later, called an d- FSA, which can be obtained algorithmically from the given grammar. The d-FSA gives the program necessary to parse the RG or RE.

General Parser

The **General Parser** parses a language defined as a string of tokens— operators, identifiers, keywords etc. to form assignment of algebraic expressions, conditionals, etc which form the computer language to be parsed. The language is usually defined with a **CFG like grammar, which can be parsed in linear time.**

The **Abstract Translation** takes the parse which comes from the General Parser and generates code in a **generic assembly language**, which can be in turn translated into any particular assembly language. For arithmetic operations a three address assembly language can serve this purpose. It is the Code Generator that turns that abstract code into code for a particular code. **The Code Generator** is the only part of the compilation that is **machine, but not source program, dependent.** All the other parts are source program, but not machine, dependent.

Optimizations can be done in either or both the Abstract Translation and Code Generation parts of the compiler.

Optimization: More Examples

In Abstract Translation (Machine Independent): Remove assignments uneffected by cycling through a loop, (right side are unchanged in the loop) from that loop. Replace multiple common sub-expression instances by a variable to which that common sub-expression is assigned. Detect Tail recursive Definition for which stacking calls is unnecessary.

<pre> x = (y+z+w) * u / (y+z) T1=y+z T2=T1+w T3=y+z T4=u/T3 x=T2*T4 </pre>	<pre> T1=y+z T2=T1+w T4=u/T1 x=T2*T4 </pre>	<pre> T1=y+z T1=T1+w T4=u/T1 x=T1*T4 </pre>
\Rightarrow		
<pre> M=0; while(i <= N) { U=100; if (x[i] > M && x[i] <= U) M=x[i]; } </pre>		
\Rightarrow		
<pre> M=0 while(i <= N) {if (x[i] > M && x[i] < U) M=x[i] } </pre>		
<pre> proc f(n) { if (n>1) f(n) = n + f(n-1); if (n=1) return(1); } F = f(5); </pre>		
\Rightarrow		
<pre> F = 1; while(n != 5) {F = n + F; n = n-1;} </pre>		

Optimization Examples

In Code Generation: (Machine dependent): Using registers for temporary storage or using special builtin machine instructions to replace a sequence of Abstract Language instructions. We have previously seen the different possibilities with 1 and 2 address machine languages.

The Lexicographic Section And Its Regular Grammar, RG

The Lexicographic section of the compiler views the input string or program as simply a **sequence of lexicographic units i.e., keywords, identifiers, numbers, operations, and punctuation (end of statement semi-colons, parenthesis, etc.)**. This view of the input can be described with limited form of CFG called an RG whose rules are restricted to two types namely:

$N \rightarrow t$, or $N \rightarrow t M$ in which t is a single TS and N and M are single NTS, perhaps the same

The input is parsed according to that RG, and through this parse identifiers, numbers, etc. are detected and translated into tokens. Identifiers are also entered into a symbol table. A token replaces each lexicographic unit of the output to the lexicographic section. A token is generally a shorthand notation which indicates the nature and, where necessary, some further detail of the unit it replaces. Thus a number can be represented in binary, an operations by a binary indication of the fact that it is an operations which of the limited number of operations it is, identifiers can be replaced by an indication that they are identifiers and a pointer to the symbol table where other pertinent information is stored for use in the Syntax Analysis section. In general the token contains some information of immediate importance in the Syntax Analysis section explicitly and further information (when necessary) in the symbol table location given by the pointer. This significantly decreases the length of the input that must be passed to the Syntax section.

The Lexicographic Section does more in parsing than check the input program for syntactic conformity to its RG. The RG parser also indicates when a lexicographic unit has been detected and what it is. For example, it detects the beginning of an identifier in the parse when one of the few rules which can serve for this purpose are used. It begins accumulating the symbols in the identifier at this point. When the end of the identifier is similarly detected the accumulated identifier is completed and ready for storage in the symbol table.

The following RG identifiers, which are any strings of **a's** and **b's**. The grammar defines arithmetic expressions involving **+**, and ***** with these identifiers.

<pre> S ----> (a b) <id> <id> ----> (a b) <id> (+ -) <op> . <op> ----> (a b) <id> </pre>	<p>IMPLEMENTED BY FSA (RG)</p>
RG	RG Parsed By an FSA-Example

In the General Parser section also the parser does more than simply recognize a program it also is used in the semantic interpretation of the program.

The Syntax Analysis Section and Operator Precedence

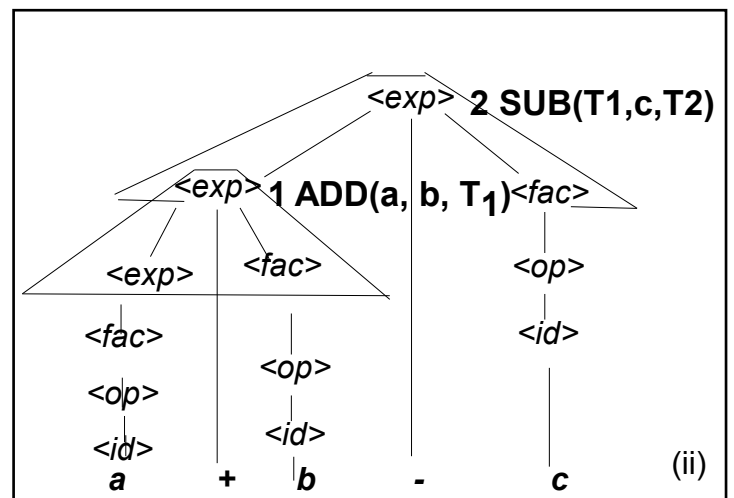
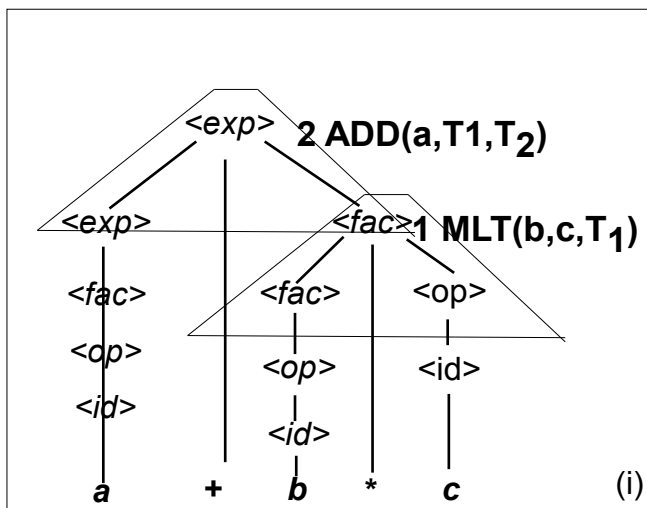
When an input string is parsed the compiler is not only able to determine whether that string is in the syntax of the language as defined by its grammar, it is also able to determine other facts necessary for translation. For example the expression $a + b * c$ requires that a multiplication be done before the addition. Using the Precedence Expression Grammar repeated here

```

<exp> ----> <fac> | <exp> + <fac> | <exp> - <fac> [ <exp>.
<fac> ----> <op> | <fac> * <op> | <fac> / <op>
<op> ----> <numb> | <id> | (<exp>)
<numb> ---> 0 | 1 | 2 | 3
<id> ----> a | b | c | d | e

```

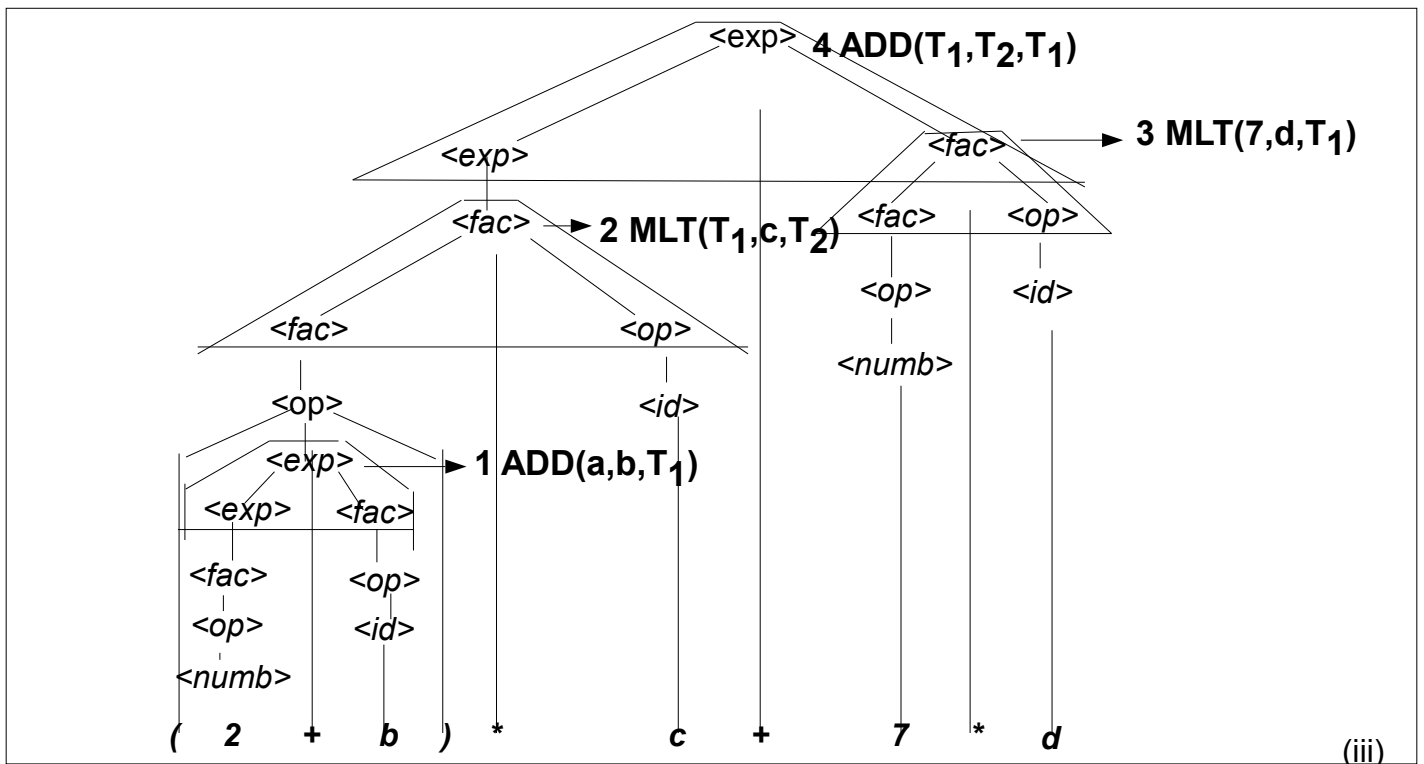
the parse tree in the figure (i), below is obtained for this expression. Notice that the subtree for the rule with * on its right appears as an ancestor of the rule with + on the right. In a bottom up parse the * rule would be recognized before the + rule. In general using this grammar, operations detected earliest in a left to right bottom up parse, should be the earliest executed. So translations to machine language for operations should be produced in this order. If this rule is followed for all parses using this grammar subexpression will be translated in the order prescribed by the familiar precedence rule. Consider the expression $a + b - c$, figure (ii), below-next page. $a + b$ should be translated first (+ and - have the same precedence but that on the left is to be translated first (left associativity).



These translations are to occur in the same order in which the subtrees are detected in a bottom up, left to right parse. In the (iii), next page, the effect of parenthesis in placing the parenthesized expression lower down on the parse tree than the operation enclosed would otherwise require is shown in a parse of $(2 + b) * c + 7 * d$

[[As described above the grammar contains the information about precedence as well as about the syntactically correct statements so that the parse can determine both. Another possible approach would be that the grammar only gave information about the correct syntax, so it could even then be ambiguous. Then the precedence information would have to be supplied by some other form of specification. This separation of syntax and precedence was popular in the past.]]

Syntax Analysis : Abstract Machine Language Generation With Precedence Grammar Examples



The **General Parser**, like the Lexicographic Unit (d-FSA) is implemented by an algorithm or machine of a standard kind. The CFG grammar involved is usually a variant of an LR(k) or LL(k) grammar and so the parsing algorithm (or machine) typically involves the use of a stack as it is used in parsing such languages.

The Abstract Translation Section

The abstract assembly language is analogous to three-address assembly code. Its use is illustrated on the previous page in **bold** print.

The Code Generation Section

If **add(a,b,c)** is a sample statement in the abstract translation and the machine is one with single address assembly code and an accumulator, then the assembly or object code for will be **cla a; add b; sto c:**

Phrase Structured Grammars; Types 3, 2, 1, and 0

There are four types of phrase structured grammars. They are arranged in order of increasing power from type 3 (*the weakest*) to type 0 (*the strongest*). A grammar of type i is also of type $i-1$. Each has a finite set of NTSs, and a finite set of TS's. For all of them derivation is defined in the same way, i.e., the result of a sequence of substitution of rule rightsides for the corresponding rule left side occurrences in sentential forms. They differ in what is allowed on the left and right of a production rule. In all of them the rule form is same:

$$\alpha \longrightarrow \beta.$$

We also give, with a phrase structure grammar of type i , ($i > 1$), a language it can describe, which cannot be described with a grammar of type $i + 1$. The differences will be given using the following abbreviations:

N and M are each any single NTS
 t = any single TS, or ϵ (0 length string)
 ω = any string of NTS's and TSs
 γ = any string of NTS's and TS's

Type 3 or Regular:

(There are two kinds.)

$N \longrightarrow t$, or $N \longrightarrow t M$ ($M \text{ can} = N$) (called Left Regular)
 or Left Regular

$N \longrightarrow t$, or $N \longrightarrow M t$ ($M \text{ can} = N$) (called Right Regular)

(A regular grammar cannot be both Left and Right regular)

Type 2 or C(ontext) F(ree) G(rammar)

$N \longrightarrow \gamma$.

$\{a^i b^j \mid i > 1\}$ also all strings of balanced parenthesis, ex. $((()((()()))))$ are type 2, not type 3 languages.

Type 1 or C(ontext) S(ensitive) G(rammar)

$\omega \longrightarrow \gamma$. with $|\omega| \leq |\gamma|$

$\{a^i b^j c^k \mid i > 1\}$ also $\omega \# \omega \mid \omega \text{ is any string from any alphabet, } S, |S| = 2\}$ are type 1, but not a type 2 language.

Type 0:

$\omega \longrightarrow \gamma$ With No Restriction

Grammars of type 0 and 1 are largely of theoretical interest. Types 2 and 3. Are widely used in specifying programming languages. The breakdown is the typical compiler design shown in figure 4 indicates where compilation is RG guided and where it is CFG guided. It will be necessary to include some features exemplified by the languages above which are definable with CSG's and not with CFG's. For this purpose the Attribute Grammar to be considered later, is normally used rather Than a CSG grammar.