

SYNTAX ANALYSIS (PARSING)

The journey in this phase of a compiler as it is stated in the syllabus.

Module II (From the Syllabus) :

Introduction to Syntax Analysis, Elimination of Ambiguity, Left Recursion and Left Factoring, Recursive and Non-Recursive Top-Down Parsers, Bottom-up Parsers: Shift Reduce Parser techniques and conflicts, all variants of LR Parsers, Handling Ambiguous grammar in Bottom- Up Parsing, Error handling while parsing, the Parser generator YACC. **(15 Lectures)**

SYNTAX ANALYSIS OR PARSING

1.Motivation

After lexical analysis, syntax analysis (or parsing) is the next phase in compiler design. Syntax analyser (or parser) is the name of that part of a compiler which performs this task.

```
main ()
{
int i,sum;
sum = 0;
for (i=1; i<=10; i++)
sum = sum + i;
printf("%d\n",sum);
}
```

| | | | | | | | | | | | | | | | | |
|------|--------|---|--------|-----|-----|----|-----|---|-----|---|---|-----|-----|-----|---|---|
| main | (|) | { | int | i | , | sum | ; | sum | = | 0 | ; | for | (| | |
| i | = | 1 | ; | i | <= | 10 | ; | i | ++ |) | ; | sum | = | sum | + | i |
| ; | printf | (| "%d\n" | , | sum |) | ; | } | | | | | | | | |

In order to check the syntactic validity of the code fragment, we need the rules for constructing the various linguistic features used there.

The given code is a definition of function named `main()`. The body of the function has declarative statement, a loop construct, an assignment statement, a function call for output, etc. Let us use our intuition and exposure to C to detail out a specification of some of these constructs in a semi formal manner.

- Function definition usually requires a `return_type` for the return value of a function if any, its name, zero or more arguments enclosed in a pair of parentheses, its body of statements enclosed in another pair of curly braces, and so on. In a symbolic manner, we shall write rules of the form :

`language_feature` \rightarrow `description`

Choosing names to describe a feature in terms of its sub features is a possible approach.

- `fundef` \rightarrow `rettype` `fname` (`arglist`) {`body`}

The name `fundef` is chosen to define the structure of a function definition in C. The terms used on the rhs of \rightarrow describe the components that form a definition. The terms `rettype`, `fname`, `arglist`, `body` are the names chosen to describe the return-type, function name, list of parameters followed by the statements that usually constitute the body of a function.

- `rettype` \rightarrow `int` | `void` | ϵ

A few common return types are specified for the present discussion. In general one would have to include all the valid return types permitted in C. The symbol ϵ denotes an empty string indicating return type is not mandatory and may be skipped.

- `fname` \rightarrow `id`

A function name is specified as any valid identifier (`id`) in C.

- The two parentheses, ‘(’ and ‘)’ are mandatory part of a function definition.
- The name `arglist` specifies the manner in which parameters are specified in C, however since our `main()` does not use them, we shall use ϵ for this feature.

`arglist` \rightarrow ϵ

A general description of `arglist` would be of the form

`arglist` \rightarrow `arglist` | ϵ
`arg` \rightarrow `type id` | `arg, type id`

- $\text{body} \rightarrow \text{decl slist} \mid \epsilon$

decl denotes a declaration statement while slist stands for a list of statements.

- $\text{decl} \rightarrow \text{type idlist}; \quad \text{idlist} \rightarrow \text{idlist, id} \mid \text{id} \quad \text{type} \rightarrow \text{int} \mid \text{float}$

The descriptions given above are adequate to generate a single declaration statement with a type followed by a list of comma separated identifiers.

Let us consolidate all the semi-formal specifications written above as follows. These are to be taken as indicative and complete specifications are significantly larger and involve more features even to specify the simple subset of C chosen here.

```

fundef  $\rightarrow$  rettype fname (arglist) {body}
rettype  $\rightarrow$  int | void |  $\epsilon$ 
fname  $\rightarrow$  id
arglist  $\rightarrow$   $\epsilon$ 
body  $\rightarrow$  decl slist |  $\epsilon$ 
decl  $\rightarrow$  type idlist;
idlist  $\rightarrow$  idlist, id | id
type  $\rightarrow$  int | float

```

The objective is to get some insight into the process of syntax checking of a code fragment. We shall use the earlier code fragment along with the specification of constructs listed above.

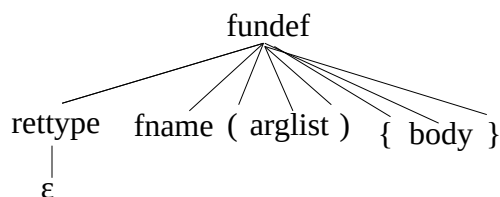
Objective : To perform a parsing or syntax analysis of the following code using the specifications given above. Let us assume that we are told that a function definition is to be analysed.

```
main () {int i, sum; sum = 0; for (i=1; i<=10; i++) sum = sum + i; printf("%d\n", sum);}
```

The current token is : main

To match with a function definition, we use the first description.

The specification $\text{fundef} \rightarrow \text{rettype fname (arglist) \{body\}}$ requires that the 8 components that it comprises of needs to be matched.



We shall use this tree structure to go ahead with intuitive parsing of the input code fragment. The tree is placed in the last column of a table that shows the progress in parsing along with the associated actions that are taken.

| Current token | Action Taken | Checking for Syntactic Structure |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| main | Since there is no return type, we use the descriptions $rettype \rightarrow \epsilon$, and proceed. | <pre> graph TD fundef --> rettype fundef --> fname fundef --> LParen["("] fundef --> arglist fundef --> RParen[")"] fundef --> LBrace["{"] fundef --> body fundef --> RBrace["}"] rettype --> epsilon["ε"] </pre> |
| main | main is an identifier, which matches with fname and the tree is extended accordingly. | <pre> graph TD fundef --> rettype fundef --> fname fundef --> LParen["("] fundef --> arglist fundef --> RParen[")"] fundef --> LBrace["{"] fundef --> body fundef --> RBrace["}"] rettype --> epsilon["ε"] fname --> main </pre> |
| (| matches the structure | Same tree. Now arglist is required |
|) | No arguments given in the input; use the description $arglist \rightarrow \epsilon$ The input has matched till arglist of the specification of a function definition | <pre> graph TD fundef --> rettype fundef --> fname fundef --> LParen["("] fundef --> arglist fundef --> RParen[")"] fundef --> LBrace["{"] fundef --> body fundef --> RBrace["}"] rettype --> epsilon1["ε"] fname --> main arglist --> epsilon2["ε"] </pre> |
|) | Matches with the specification. Get the next token. | Same Tree. |
| { | Matches with specification. Get the next token. | Same Tree. Now the structure of body is expected to be seen in the input. |
| int i, sum; | Proceeding along the same lines, what would the tree structure after ';' has been read from the input | <p>The tree is drawn below.</p> <p>The nonterminal slist remains to be explored with the remaining input :</p> <pre>sum = 0;for (i=1; i<=10; i++)sum = sum + i;printf("%d\n",sum);}</pre> |

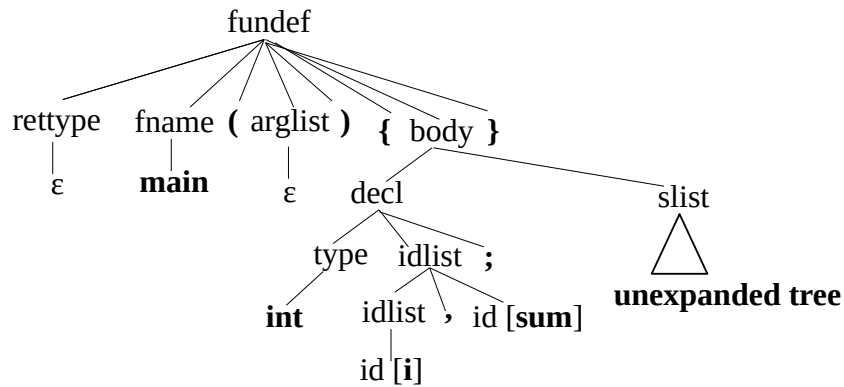


Figure : Tree after partially processing the input : `main () {int i, sum; sum = 0;}`

The preceding material introduces the essential issues in parsing while skipping the details. The terms and basic concepts of parsing are addressed in the rest of the document.

Basic Issues in Parsing

Q. What does a parser (or syntax analyzer) do?

- groups tokens appearing in the input and attempts to identify larger structures in the program. This amounts to performing a syntax check of the program.
- makes explicit the hierarchical structure of the token stream. Associated is the issue of representation - how should the syntactic structure of a program be explicitly captured ? This information is normally required by subsequent phases.

Q. How to specify Programming Language (PL)Syntax ?

It is obvious that a parser must be provided with a description of PL syntax. How to describe the same is an important question and the related issues are :

1. The specification be precise and unambiguous.
2. The specification be complete, that is cover all the syntactic details of the language.
3. Specification be such that it be a convenient vehicle for both the language designer and the implementer.

We have already studied a formalism called Context Free Grammar in the course on “Formal Languages and Automata”. In this module we shall examine the extent to which CFG meets the requirements stated above in order to apply it to parsing.

Q. How to represent the input after it has been parsed ?

We shall study one representation known as parse tree in this context. The same representation has many variants, such as Abstract Syntax Tree (AST) and others.

Q. What are the parsing algorithms, how they work and what are their strengths / limitations ?

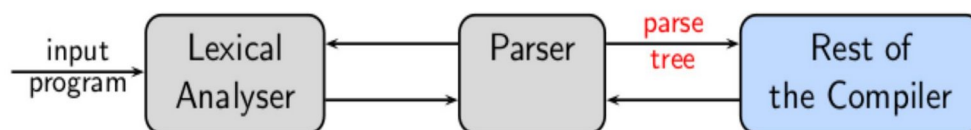
This is the primary concern of the module. We shall discuss two different approaches to parsing , known as, top-down parsing and bottom- up parsing, and study some parsing algorithms of both types. The two parsers are not equivalent in their applicability and the strengths of parser in the family of parsers shall be discussed here.

- In syntax analysis phase, we are not interested in determining what the program does (known as the semantics of the program) and hence such issues are not addressed here. The semantic analysis aspects are discussed separately.

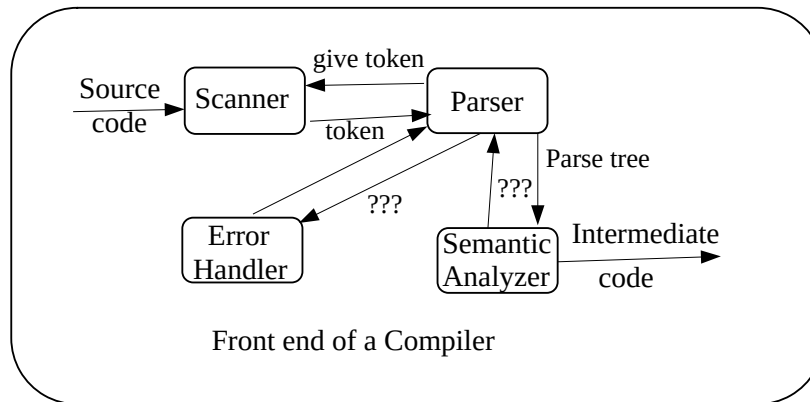
Q. How are parsers constructed ?

- Till about 2-3 decades back, parsers (in fact the entire compiler) was written manually.
- A better understanding of the parsing algorithms has led to the development of special tools which can automatically generate a parser for a given PL.
- As we shall see later both top-down and bottom-up parsers can be automatically generated.
- A compiler generation tool, called YACC (Yet Another Compiler Compiler), generates a bottom-up parser and is available on Unix. Another such tool is BISON available in the GNU public domain software.
- Similarly ANTLR (ANother Tool for Language Recognition), is a parser generator tool that generates top down parsers. However top down parser generators are outside the scope of this course.

Interaction of Parser with scanner and the rest of the compiler modules was shown earlier as depicted in the following figure.



Let us extend the earlier figure to include the next phase of the compiler known as semantic analyzer.



The legend “???” indicates that the exact interface between the modules depends on the concerned modules. It should be noted that though syntax and semantic analysis are two distinct phases of a compiler because they perform different functions, that are often integrated into the same pass of the compiler in practice.

Specification of PL Syntax

Program in any language is essentially a string of characters from its alphabet; however an arbitrary string is not necessarily a valid program.

- The problem of specifying syntax is to determine those strings of characters that represent valid programs (programs that are syntactically correct). Rules which identify such valid programs spell out the syntax of the language.
- Recall that tokens are the lexical structures of a language and can be defined precisely and formally. This helps in the automatic generation of lexical analyzers.

What is the situation with syntactic structures and parsing ?

- Syntactic structures are grouping of tokens. The language definition provides the syntactic structures that are permitted. A few syntactic structures common across PLs are

| | | | |
|--------------------|-------------|-----------|-------|
| expression | declaration | statement | block |
| compound-statement | function | program | |

- Syntax of expressions is nontrivial - depends on the richness and the properties of the operators supported in the language. The operator set of C / C++, that has been shared with you, is a glaring example of this fact.

- Since you are quite familiar with the syntax of C, let us use a different PL known as Pascal to informally describe a possible syntax for **variable declarations** in Pascal (was very popular in the 70s but not used much today)

- a stream of tokens that has keyword **var** as the first token
- followed by one or more tokens of type **identifier**, separated by token **comma**
- followed by token **colon**
- followed by any one of the tokens, **integer** or **real**
- followed by token **semicolon**

An example of a variable declaration in Pascal is : **var a,b,c : integer;**

Contrast this to an equivalent declaration in C : **int a, b, c;**

Even for such a simple syntactic structure, the problems with informal description (the second rule specially) are evident. This is the motivation for formal specification of syntax.

To summarize, the following questions address some of the basic issues related to specification of syntax that a parser writer must be aware of.

A few relevant questions in this context.

Q1. Does there exist formalism that can be used to describe all the syntactic details of a PL ? If the answer is yes, then can the same formalism be used to write a parser for the language ?

Q2. If one uses a formalism that captures most of the syntactic details (but not all) of a PL and uses the same to write a parser, then when and how are the missing features taken care of ?

Q3. Given a formal mechanism , can the syntax of a language be specified uniquely ?

Q4. If the answer to above is no, then which among the several specifications should be preferred while constructing parsers and why ?

Q5. Is it useful for a compiler writer to know how to write syntactic specification that is suitable for parsing ?

Basic Concepts in Parsing : Context Free Grammar

We introduce a notation called Context Free Grammar (CFG) informally and then give a formal definition.

- It is a notation for specifying programming language syntax.
- Identify the syntactic constructs of the language , such as expression, statement, etc. and express its syntax by means of rewriting rules. We continue with sample specifications of a declaration statement in Pascal and C, as given below. The

| Grammar for Single Declaration in Pascal | Grammar for Single Declaration in C |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| declaration \rightarrow var decl-list decl-list \rightarrow list : type; list \rightarrow id list , id type \rightarrow integer real | declaration \rightarrow type decl-list ; decl-list \rightarrow id list , id type \rightarrow int float |

The symbols that appear in an input are marked in bold in the table above. Concatenation is the assumed operator between the symbols in the right hand side (rhs).

- Apart from tokens (also called as **terminals**), other symbols have been used in the **rewrite rules** above. These symbols called **nonterminals** (or syntactic category) do not exist in a source program.
- A nonterminal symbol only is allowed to appear in the left hand side (lhs) of a rewrite rule. However in the rhs, both nonterminal and terminal symbols may appear.
- The rhs of a rewrite rule (also called a **production rule**) specifies the syntax of the lhs nonterminal. It may contain both kinds of symbols.
- Since the issue is to parse (and compile) programs, a nonterminal that specifies the syntax for a program must be present in the CFG. For instance a C program may be defined as a collection of one or more function definitions preceded by zero or more global declarations. The nonterminal program is called the **start symbol**. The nonterminals in the following are marked in *italics*, separate them from the terminal symbols.

program \rightarrow *global-decls* *func-list*
func-list \rightarrow *func* | *func func-list*
func \rightarrow *ret-type func-name (arglist) { func-body }*
...

Example of the structure of a program in pascal :

program \rightarrow **program id** (*list*) ; *decls* *compound-statement*

- Other nonterminals are introduced to specify the various parts of the start symbol, such as **program** above, in a structured manner.

A CFG is formally defined in the following . It has four components, G is the grammar, and written as $G = (T, N, S, P)$, where

- T is a finite set of terminals (or tokens)
- N is a finite set of nonterminals
- S is a special nonterminal (from N) called the start symbol
- P is a finite set of production rules of the form

$$A \rightarrow \alpha, \text{ where } A \text{ is from } N \text{ and } \alpha \text{ is from } (N \cup T)^*$$

There can be more than one definition for a nonterminal (different rhs for the same lhs) in which case the definitions are separated by the symbol ‘|’.

Q. Why the term context free ?

1. $\text{left_symbol} \rightarrow \text{right_symbols}$ is the only kind of rule permitted in a CFG; where left_symbol is a single nonterminal and right_symbols is a string from $(N \cup T)^*$
2. Rules are used to replace an occurrence of the lhs nonterminal by its rhs. In a CFG, the replacement is made regardless of the context of the lhs nonterminal (symbols surrounding it).

Grammars, as we know from our experience with languages (natural or PL), are used to define languages.

Q. What is the relation between CFG of a PL and the PL itself ?

1. Intuitively, a PL may be defined by the collection of all valid programs that can be written in that language.
2. A grammar is used to define a language in the sense that it provides a means of generating all its valid programs.
3. Beginning with the start symbol of the grammar and using production rules repeatedly (for replacing nonterminal symbols), one can produce a string comprising terminals only. Such sequence of replacements is called a derivation.
4. The set of all possible terminal strings (elements of T^*) that can be derived from the start symbol of a CFG G is an important collection. Informally this set of strings is the language denoted by G .

Example : Consider a CFG , $G = (T, N, S, P)$ with $N = \{list\}$, $S = list$, $T = \{id, , \}$ and P containing 2 rules, $P :$

1. $list \rightarrow list, id$
2. $list \rightarrow id$

which could also be combined to the compact form $list \rightarrow list, id \mid id$

A derivation is traced out as follows

| | | |
|------|---------|--------------|
| list | derives | list, id |
| | derives | list, id |
| | derives | list, id, id |
| | derives | id, id, id |

The derivation process stops when the string “**id , id , id**” is reached because this string does not any nonterminals. Using the derivation process, we can see that grammar G is capable of generating terminal strings such as

$L(G) = \{id\ id, id\ id, id, id\ id\ \dots\}$.

In English, we may describe the language denoted by G , that is $L(G)$ as the non-empty set of strings that comprise of one or more **id** separated by commas (,).

Notational Conventions

Since we have to frequently refer to terminals and nonterminals, the following conventions are commonly used in the literature on CFG and Parsers.

| Symbol type | Convention |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| single terminal | small case letters from the start of the English Alphabet, such as {a, b, c,...}, operators, delimiters, keywords, etc. |
| single nonterminal | upper case letters from the start of the English Alphabet, such as {A, B, C,...}, names such as <i>declaration</i> , <i>list</i> , <i>expression</i> , etc. <i>S</i> is typically reserved for the start symbol. |
| single grammar symbol | upper case letters at the end of the English Alphabet, such as {X, Y, Z,...} |
| string of terminals | small case letters at the end of the English Alphabet, such as {x, y, z,...} |
| string of grammar symbols | Greek letters such as { α , β , γ , δ , ...} |
| null string | ϵ |

Formal Definitions

The concepts and terms that have been introduced informally so far, are now formally defined.

Let $A \rightarrow \gamma$ be a production rule.

Consider a string $\alpha A \beta$ from $(N \cup T)^*$. Using the convention stated above, the string $\alpha A \beta$ indicates that there is a nonterminal A in the string which is surrounded on either side by some grammar symbols. The key idea for choosing such a string is to notify the presence of a nonterminal A in it.

1. Replacing the nonterminal A , by its definition $A \rightarrow \gamma$ in the string $\alpha A \beta$ above, yields the new string $\alpha \gamma \beta$. Formally this is stated as $\alpha A \beta$ derives $\alpha \gamma \beta$ in a derivation of one step. A concise form of writing a **one step derivation** is :

$\alpha A \beta \Rightarrow \alpha \gamma \beta$, where the symbol \Rightarrow stands for **derives in one step**.

2. If $\alpha_1, \alpha_2, \dots, \alpha_n$ are arbitrary strings of grammar symbols, such that $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \dots \Rightarrow \alpha_n$, then we say that α_1 derives α_n .

3. If a derivation comprises of zero or more steps, the symbol \Rightarrow^* is used. Clearly for any string α , it is obvious that $\alpha \Rightarrow^* \alpha$ is true.

4. If a derivation comprises of one or more steps, the symbol \Rightarrow^+ is used.

5. It is interesting to note that $\Rightarrow, \Rightarrow^*$ and \Rightarrow^+ are relations over $(N \cup T)^*$ in the set theoretic sense of relation; where \Rightarrow^* is the reflexive transitive closure of \Rightarrow , while \Rightarrow^+ is the transitive closure of \Rightarrow . The last of these three relations is used to define the concept of a language.

6. The language $L(G)$ denoted by a context free grammar G is defined by $L(G) = \{ w \mid S \Rightarrow^+ w, w \in T^* \}$. Such a string w is called a **sentence** in $L(G)$.

7. A string $\alpha, \alpha \in (N \cup T)^*$, such that $S \Rightarrow^* \alpha$, is called as a sentential form in $L(G)$.

8. Grammars $G_1 = (T, N_1, S_1, P_1)$ and Grammars $G_2 = (T, N_2, S_2, P_2)$ are said to be equivalent, written as $G_1 \equiv G_2$, if they generate the same language, i. e., $L(G_1) = L(G_2)$.

EXAMPLE 1: Illustrations of Derivations and sentential forms

Consider the 3 grammars given below.

$G_1 = (T, N_1, L, P_1)$, where $T = \{ , id \}$; $N_1 = \{ L \}$ and $P_1 = \{ L \rightarrow L , id \mid id \}$.

$G_2 = (T, N_2, L, P_2)$, where $N_2 = \{ L \}$ and $P_2 = \{ L \rightarrow id , L \mid id \}$

$G_3 = (T, N_3, L, P_1)$, where $N_3 = \{ L, L' \}$ and P_3 has 3 rules given below.

$$\begin{aligned} L &\rightarrow id L' \\ L' &\rightarrow , id L' \mid \epsilon \end{aligned}$$

It can be shown that the set of strings generated by all the grammars is $\{id \ id, id \ id, id, id \ \dots\dots\dots\}$.
Therefore since $L(G_1) = L(G_2) = L(G_3)$, all the three grammars are equivalent, or $G_1 \equiv G_2 \equiv G_3$.

Consider grammar G_1 and construct a derivation of the string “id , id, id” from the start symbol L .

$$L \Rightarrow L , id \Rightarrow L , id , id \Rightarrow id , id , id$$

The derivation involves 4 sentential forms $\{L \ L , id \ L, id, id \ id, id, id\}$ in the order they are generated of which the last one is also a sentence.

What is the type of this derivation among {leftmost, rightmost, arbitrary}?

The derivation of the string “id , id, id” using grammar G_2 turns out to be the following.

$$L \Rightarrow id , L \Rightarrow id , id , L \Rightarrow id , id , id$$

The derivation above involves 4 sentential forms $\{L \ id, L \ id, id, L \ id, id, id\}$ in the order they are generated with the last sentential form also being a sentence.

What is the type of this derivation among {leftmost, rightmost, arbitrary}?

The derivation of the string “id , id, id” using grammar G_3 is given by the following.

$$L \Rightarrow id L' \Rightarrow id , id L' \Rightarrow id , id , id L' \Rightarrow id , id , id$$

This derivation involves 5 sentential forms $\{L \ id L' \ id, id L' \ id, id, id L' \ id, id, id\}$ in the order generated with the last sentential form also being a sentence. What is the type of this derivation among {leftmost, rightmost, arbitrary}?

Derivations And Their Use

1. A derivation of w from S is a proof that w is in the language of G , or $L \in L(G)$.
2. Derivation provides a means for generating the sentences of $L(G)$.
3. For constructing a derivation from S , there are options at two levels
 - choice of a nonterminal to be replaced among several others
 - choice of which particular rule (since there could be more than one rules for a given nonterminal) corresponding to the nonterminal selected.
4. Instead of choosing the nonterminal to be replaced during derivation, in a given sentential form, in an arbitrary fashion, it is possible to make an uniform choice at each step. Two natural selections for the replacement of a nonterminal are
 - replace the leftmost nonterminal in a sentential form
 - replace the rightmost nonterminal in a sentential form

The corresponding derivations are known as leftmost and rightmost derivations respectively.

5. Given a sentence w of a grammar G , there may be several distinct derivations for w .

EXAMPLE 2 : Consider the following expression grammar, the name of the grammar is pertinent, since it generates arithmetic expressions involving 5 terminal symbols $\{ \text{id } () + * \}$. We shall write a CFG by specifying the production rules henceforth, since the other components can be inferred from the context.

| CFG for expressions | Using the Grammar for Derivations |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E + T \mid T$ | The grammar has 6 production rules. The start symbol is nonterminal E . There are 3 nonterminals $\{E, T, F\}$. There are 5 terminals $\{ \text{id } () + * \}$ |
| $T \rightarrow T * F \mid F$ | |
| $F \rightarrow (E) \mid \text{id}$ | |

- Let us derive the sentence **id** using the production rules. We shall mark the nonterminal, that is chosen for replacement, by double underline in the sentential form. It may be noted the derivation of **id** from E took 3 steps and there was no choice for selection of a nonterminal as each sentential form had a single nonterminal.

$$\underline{\underline{E}} \Rightarrow \underline{\underline{T}} \Rightarrow \underline{\underline{F}} \Rightarrow \mathbf{id}$$

However, there was a choice for selecting one of the two alternates for each of E, T and F. For example, one could have the other alternative rule for E in the first step resulting in

$$\underline{\underline{E}} \Rightarrow E + T$$

Unfortunately from the sentential form “E + T”, it is not possible to derive **id**, regardless of which nonterminal was chosen for replacement.

- Let us now derive the sentence “**id + id * id**”

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow E + \underline{\underline{T}} \Rightarrow E + \underline{\underline{T}} * F \Rightarrow E + \underline{\underline{F}} * F \Rightarrow \underline{\underline{E}} + id * F \Rightarrow \underline{\underline{T}} + id * F \\ &\Rightarrow \underline{\underline{F}} + id * F \Rightarrow id + id * \underline{\underline{F}} \Rightarrow id + id * id \end{aligned}$$

In the derivation above, the nonterminals chosen for expansion in the 2nd sentential form onwards are {last, middle, middle, first, first, first, last} and do not have any fixed pattern. Such a derivation will be referred to as an arbitrary derivation.

- Consider the following derivation of the same sentence “**id + id * id**”

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow E + \underline{\underline{T}} \Rightarrow E + T * \underline{\underline{F}} \Rightarrow E + \underline{\underline{T}} * id \Rightarrow E + \underline{\underline{F}} * id \Rightarrow \underline{\underline{E}} + id * id \\ &\Rightarrow \underline{\underline{T}} + id * id \Rightarrow \underline{\underline{F}} + id * id \Rightarrow id + id * id \end{aligned}$$

In the derivation above, the nonterminals chosen for expansion is always the last nonterminal (or the right-most nonterminal) in all the sentential forms. Such a derivation is referred to as a rightmost derivation.

- A leftmost derivation of the same sentence will be the one in which the leftmost nonterminal is selected for replacement in every sentential form. The leftmost derivation is shown below.

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow \underline{\underline{E}} + T \Rightarrow \underline{\underline{T}} + T \Rightarrow \underline{\underline{F}} + T \Rightarrow id + \underline{\underline{T}} \Rightarrow id + T * \underline{\underline{F}} \\ &\Rightarrow id + \underline{\underline{T}} * id \Rightarrow id + \underline{\underline{F}} * id \Rightarrow id + id * id \end{aligned}$$

The two special derivations are usually marked by adding the label **rm** or **lm** to the derivation symbol \Rightarrow , depending on whether the derivation is rightmost or leftmost. The derivations are rewritten below with the labels as stated above.

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow_{rm} E + \underline{\underline{T}} \Rightarrow_{rm} E + T * \underline{\underline{F}} \Rightarrow_{rm} E + \underline{\underline{T}} * id \Rightarrow_{rm} E + \underline{\underline{F}} * id \Rightarrow_{rm} \underline{\underline{E}} + id * id \\ &\Rightarrow_{rm} \underline{\underline{T}} + id * id \Rightarrow_{rm} \underline{\underline{F}} + id * id \Rightarrow_{rm} id + id * id \\ \underline{\underline{E}} &\Rightarrow_{lm} \underline{\underline{E}} + T \Rightarrow_{lm} \underline{\underline{T}} + T \Rightarrow_{lm} \underline{\underline{F}} + T \Rightarrow_{lm} id + \underline{\underline{T}} \Rightarrow_{lm} id + T * \underline{\underline{F}} \\ &\Rightarrow_{lm} id + \underline{\underline{T}} * id \Rightarrow_{lm} id + \underline{\underline{F}} * id \Rightarrow_{lm} id + id * id \end{aligned}$$

Derivations And Parse Trees

There is an equivalent form of depicting a derivation that is pictorial in nature and is called a parse tree. A parse tree for a context free grammar, is a tree having the following properties :

1. root of the tree is labeled with the start symbol S ,
2. each leaf node is labeled by a token or by ϵ ,
3. an internal node of the tree is labeled by a nonterminal,
4. if an internal node has A as its label, then the children of this node from left to right are labeled with $X_1, X_2, X_3, \dots, X_n$ when there is a production of the rule
$$A \rightarrow X_1 X_2 X_3 \dots X_n$$
where X_i is a grammar symbol.
5. the leaves of the tree read from left to right give the yield of the tree; essentially the sentence generated or derived from the root.

An example of a parse tree is given later.

Q. How to construct a parse tree from a derivation ?

Let $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = z$ be a derivation of sentence z from S in derivation steps.

The corresponding parse tree may be constructed as follows

- create a root labeled with S
- for each sentential form, α_i , $i \geq 1$, construct a parse tree with the yield of α_i
- If the tree for α_{i-1} is available, the tree for α_i is easily constructed (by using induction), as given below.

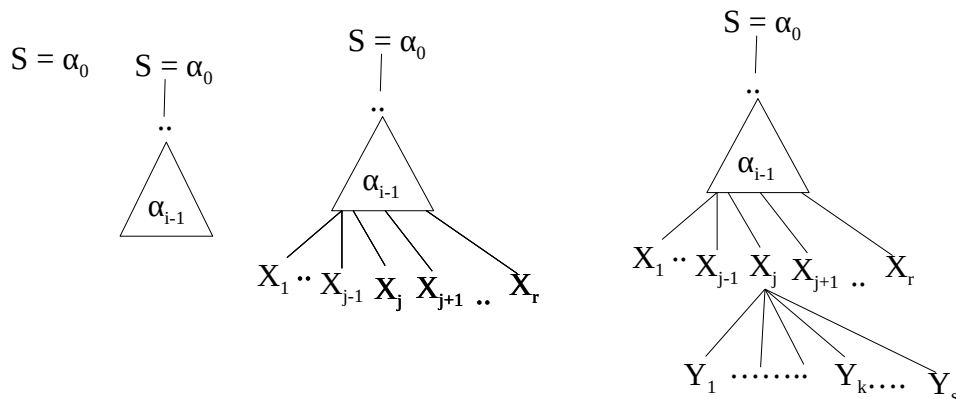
Let $\alpha_{i-1} = X_1 X_2 X_3 \dots X_r$

$\Rightarrow X_1 X_2 X_3 \dots X_{j-1} \beta X_{j+1} \dots X_r = \alpha_i$

where $X_j \rightarrow \beta$ is the rule used and let β be $Y_1 Y_2 \dots Y_s$

- The node (leaf) corresponding to X_j in the parse tree is expanded by
 - creating s number of children for the node corresponding to X_j , and
 - labeling these children with $Y_1 Y_2 \dots Y_s$ in the order from left to right.

The creation is progressively shown in the following figure. The first node is the root of the tree. The 2nd tree assumes that the tree has been drawn till the sentential form $S = \alpha_0 \Rightarrow^* \alpha_{i-1}$; the 3rd tree shows tree rooted at α_{i-1} ; with its r children labeled as X_i , $1 \leq i \leq r$ and the final tree shows the situation corresponding to the sentential form : $S = \alpha_0 \Rightarrow^* \alpha_i$



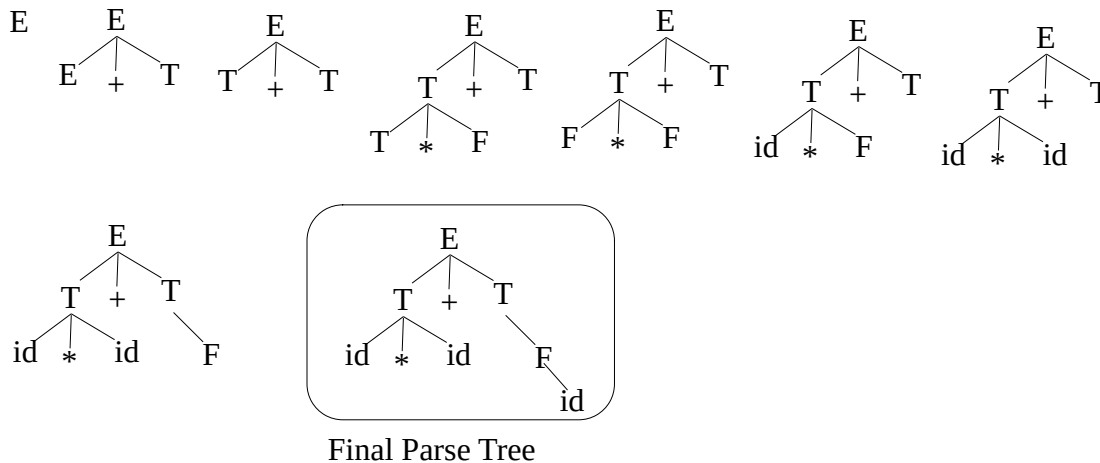
EXAMPLE : Consider the following expression grammar.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Leftmost derivation of $\text{id} * \text{id} + \text{id}$

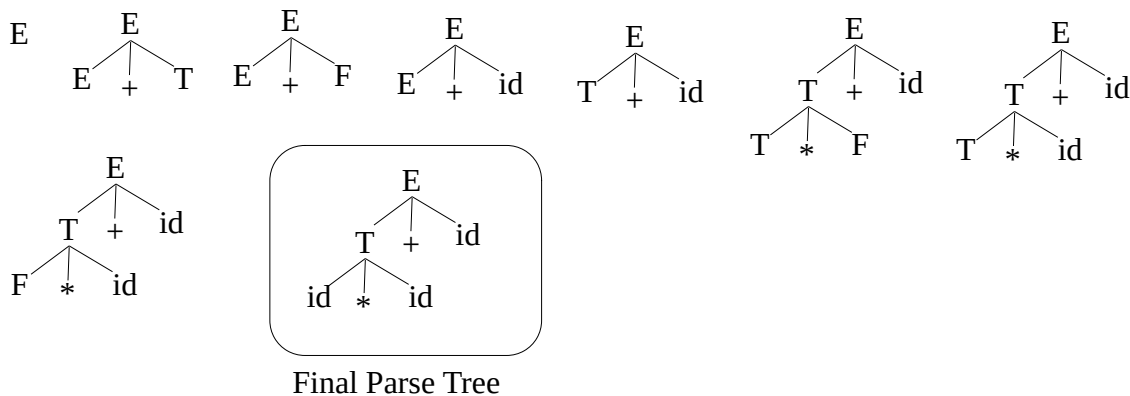
$$\begin{aligned} \underline{E} &\Rightarrow_{\text{lm}} \underline{E} + T \Rightarrow_{\text{lm}} \underline{T} + T \Rightarrow_{\text{lm}} \underline{T} * F + T \Rightarrow_{\text{lm}} \underline{F} * F + T \Rightarrow_{\text{lm}} \text{id} * \underline{F} + T \\ &\Rightarrow_{\text{lm}} \text{id} * \text{id} + \underline{T} \Rightarrow_{\text{lm}} \text{id} * \text{id} + \underline{F} \Rightarrow \text{id} * \text{id} + \text{id} \end{aligned}$$

The parse trees as the derivation proceeds are shown below.



Rightmost derivation of $\text{id} * \text{id} + \text{id}$

$$\begin{aligned} \underline{E} &\Rightarrow_{\text{rm}} E + \underline{T} \Rightarrow_{\text{rm}} E + \underline{F} \Rightarrow_{\text{rm}} \underline{E} + \text{id} \Rightarrow_{\text{rm}} \underline{T} + \text{id} \\ &\Rightarrow_{\text{rm}} T * \underline{F} + \text{id} \Rightarrow_{\text{rm}} \underline{T} * \text{id} + \text{id} \Rightarrow_{\text{rm}} \underline{F} * \text{id} + \text{id} \Rightarrow_{\text{rm}} \text{id} * \text{id} + \text{id} \end{aligned}$$



The parse tree corresponding to either of the two derivations produces the same parse tree. The tree is given below. The tree shows that its independent of the derivation that produces it, however this fact holds only when there exists a unique derivation of the sentence (both leftmost and rightmost).

Derivations And Parse Trees

The following summarize some interesting relations between the two concepts.

1. Parse tree filters out the choice of replacements made in the sentential forms.
2. Given a derivation for a sentence, one can construct a parse tree for the sentence. In fact one can draw the parse tree as the derivation proceed.
3. Even while several distinct derivations may exist for a given sentence, they usually correspond to a single parse tree.
4. Given a parse tree for a sentence, it is possible to construct a unique leftmost and a unique rightmost derivation.

Q. Can a sentence have more than one distinct parse trees corresponding to it ? Construct examples, if such is possible.

Ambiguous Grammar

A context free grammar G that produces more than one parse tree for a sentence of $L(G)$ is defined to be an ambiguous grammar.

Equivalently a context free grammar that has two or more leftmost (or rightmost) derivations for a sentence is an ambiguous grammar.

Ambiguous grammars are usually not suitable for parsing, because of the following reasons.

1. A parse tree would be used subsequently for semantic analysis; more than one parse tree would imply several interpretations and there is no obvious way of preferring one over another.
2. How does one detect that a given context free grammar is indeed ambiguous ?
3. Since multiple parse trees, even for a single sentence, renders the grammar ambiguous, is an algorithmic solution feasible ?
4. What can be done with ambiguous grammars in the context of parsing?
 - Rewrite the grammar such that it becomes unambiguous.
 - Use the grammar but supply disambiguating rules (used by the tool YACC).

Writing A CFG For PL Features

Given a PL feature, one may write several distinct but equivalent context free grammars for describing its syntax.

There are several thumb rules that can be meaningfully used in writing a grammar that is better suited for parsing.

- Many syntactic features of a language are expressed using recursive rules. Usually these can be written in either left recursive or right recursive form. While both forms may be equivalent in expressiveness, they have different implications in parsing. As an example, consider syntax for declarations :

Using a left recursive rule (the first symbol in the rhs of a rule is the same nonterminal as that in the lhs) such as the rule given below.

$D \rightarrow \text{var list} : \text{type} ;$
 $\text{type} \rightarrow \text{integer} \mid \text{real}$
 $\text{list} \rightarrow \text{list} , \text{id} \mid \text{id}$

Using a right recursive rule (the last symbol in the rhs of a rule is the same nonterminal as that in the lhs) such as

$D \rightarrow \text{var list} : \text{type} ;$
 $\text{type} \rightarrow \text{integer} \mid \text{real}$
 $\text{list} \rightarrow \text{id} , \text{list} \mid \text{id}$

Both the grammars are equivalent in terms of expressiveness, the underlying languages which can contain potentially infinite number of id, are same for both grammars.2. Writing a grammar for expressions; points of concern here are the operators and their properties.

Associative property: This property is useful to answer the question, “ In the absence of parenthesis, how an expression such as $a \theta b \theta c$, is to be evaluated” ? Here θ is some binary operation.

- An operator is classified as either **left** or **right** associative, depending upon whether an operand with same θ on both its sides is to be consumed by the θ placed to its left or right. Operators $+$, $-$, $*$ and $/$ are left associative while \uparrow (exponentiation in Pascal) or the assignment operator '=' in C are right associative.
- To honor the left(or right) associative properties of operators, left (or right) recursive rules are useful.

Precedence Property : Several operators have different precedence attached to them and an expression containing them has to be evaluated in accordance to the precedence values, else different interpretations may arise. Programming Language definition specifies the relative precedence of its permissible operators.

- Typically the operators such as $\{ *, / \}$ have higher precedence than $\{ +, - \}$; in the presence of these operators with varying precedence values, grammar rules for writing the syntax of expressions that involve these 2 sets of operators is constructed as follows.
 - A nonterminal for each of the two sets are chosen, say *term* and *exp* respectively.
 - The rules for the operators with lower precedence are :
 $\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$
Notice the use of left recursion and the use of 2 nonterminals.
 - To write the rules for the other operator class, $\{ *, / \}$ the basic units in expressions are needed. Another nonterminal, say, *factor*, is chosen for the purpose.
 $\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

- Finally, The rules for the basic unit are written :
factor \rightarrow id | (exp)
- Provide intuitive reasoning as to why the above empirical rules are adequate.

Disambiguating a Grammar : There are several useful transformations that make a CFG more amenable for parsing. Disambiguating a grammar is one of them.

- The following grammar for expressions is ambiguous, but very concise. Note that there are two instance of operator $-$, one is unary minus and the other one is a binary minus.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \text{id}$$

- Prove the fact that the CFG given above is ambiguous
- Using the empirical rules stated in item 2 above, and along with the properties of all the involved operators, we know how to write an equivalent CFG that is unambiguous.
- Consider another grammar given below. This grammar is supposed to generate nested if-then-else statements in PLs. A generic form of this conditional statement is given here, languages may have a small variation in the syntax for this construct.

$$S \rightarrow \text{if } \mathbf{c} \text{ then } S \\ \mid \text{if } \mathbf{c} \text{ then } S \text{ else } S \mid \text{other}$$

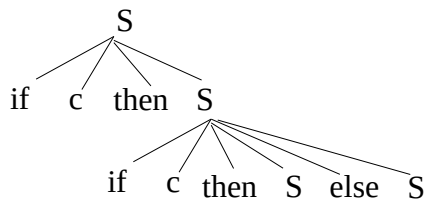
The keywords are marked in bold, the other symbols are nonterminals. We shall treat c as a terminal in this context since the focus is on conditionals. We can ignore the nonterminal, *other*, because that is supposed to take care of statements that are not conditionals.

- Is the grammar above ambiguous? Consider the sentence : “if c then if c then S else S ” and parse this sentence using the grammar for conditionals.

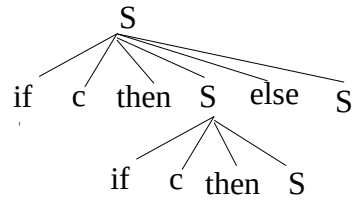
$$S \Rightarrow \text{if } \mathbf{c} \text{ then } S \Rightarrow \text{if } \mathbf{c} \text{ then if } \mathbf{c} \text{ then } S \text{ else } S \quad (1)$$

$$S \Rightarrow \text{if } \mathbf{c} \text{ then } \underline{S} \text{ else } S \Rightarrow \text{if } \mathbf{c} \text{ then if } \mathbf{c} \text{ then } S \text{ else } S \quad (2)$$

Two distinct leftmost derivations are seen above, whereby the the grammar given above is proved to be ambiguous. The same observation can also be observed if parse trees are drawn for these two derivations. The two parse trees are very different showing clearly the ambiguity.



Parse Tree for (1)



Parse Tree for (2)

Q. Is it possible to write an unambiguous grammar for conditional statements ?

In order to find an answer to the question, it is insightful to examine a sentence comprising of nested conditional statements and determine how PLs decipher the same.

Let us examine the following input for the purpose :

if then if then if then else if then else else

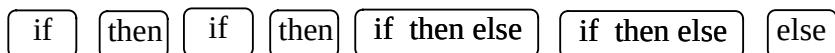
The pairing of **else** with a preceding **then** in the above sentence in PLs is done as follows :

if then if then if then else if then else else

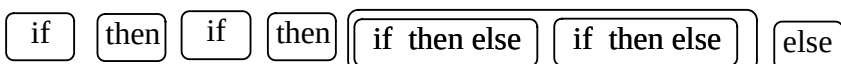
The pairing of the first else, in a left to right scan of the sentence, to its matching then, is shown as a larger box below.



The pairing of the second else to its matching then, is shown as the 2nd larger box.



The pairing of the last else to its match, is shown as the single large box, which encloses the earlier two boxes.



The pairing shown above is the unique interpretation for the given sentence. PL designers agree to the above interpretation. The matching rule can be specified as, “ in a left to right scan of the sentence, **else**

has to be paired with the closest preceding unmatched **then**.

The challenge now is to write a grammar that ensures the matching rule stated above.

Examination of the matching rule reveals that between any two consecutive **then** and **else**, only a complete “**if then else**” can occur, if any.

This crucial observation can be incorporated into a grammar that removes the ambiguity inherent in the earlier grammar.

| | |
|----------------|------------------------------------------------------------------------------------|
| S | → matched-stmt unmatched-stmt |
| matched-stmt | → if c then matched-stmt else matched-stmt other |
| unmatched-stmt | → if c then S if c then matched-stmt else unmatched-stmt |

Argue that the grammar for conditionals given above is indeed unambiguous.

Left recursion removal

We have seen that PL grammars usually have some recursive rules. We shall see later that left recursive grammars are unsuited for a class of parsing methods (top down parsing).

- A context free grammar is left recursive if there exists a nonterminal A, such that $A \Rightarrow^+ A\alpha$, for some α in $(N \cup T)^*$.

Simple case : Consider the rule $A \rightarrow A\alpha \mid \beta$. The grammar generates strings $\{\beta, \beta\alpha, \beta\alpha\alpha, \dots\}$. An equivalent grammar without left recursion is

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

It can be easily seen that the grammar above generates the same set of strings.

- If the left recursion is direct (or immediate), then even in a more general case such as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

The rules above have m left recursive rules for A and also there are n alternatives of A that do not involve recursion. A left recursion free equivalent grammar is

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$

The proof of equivalence is left to the reader.

The process of left recursion elimination does not work for all left recursive grammars. Write a few examples of left recursive grammars for which the suggested method does not work. Find the constraints under which the method will always work correctly.

Introduction To Parsing : Parsing Strategies

Let us begin with a formal definition of a parser.

A parser for a context free grammar G is a program P that when invoked with a string w as input, indicates that either

1. w is a sentence of G , i. e., $w \in L(G)$ (and may also give a parse tree for w), or
2. gives an error message stating that w is not a sentence (and may provide some information at or around the point of error).

Parsing Strategies

The two parsing strategies that we study are based on the following principles.

1. A parser scans an input token stream , from left to right, and groups tokens with the purpose of identifying a derivation, if one such exists.
2. In order to trace out such an unknown derivation, a parser makes use of the production rules of the grammar.
3. Different parsing approaches differ in the choice of derivation and/or the manner they construct a derivation for the input.
4. We have seen two standard derivations, viz., leftmost and rightmost, and also the way these can be represented through a parse tree. The two basic approaches to parsing can be easily explained using the concept of parse tree.
5. What are the possible ways in which a parse tree (or a tree data structure, in general) can be constructed?
 - Create the parse tree from the root downwards and expand the nodes till all leaves are reached, i. e., in a top down manner. Parsers of this type are known as **top-down** parsers.
 - Create the parse tree from leaves upwards to the root, i. e., in a bottom up fashion. Parsers which use this strategy belong to the family of **bottom-up** parsers.

6. There being a direct relation between parse trees and derivations, parsers which use anyone of the two parsing strategies can also be rephrased in terms of derivations.

We now describe in brief each parsing strategy in terms of its

- Basic principles : the derivation on which it is based, how the derivation is constructed, relevant issues and problems for designing a deterministic parser of this family.
- Manual construction of a parser : one or more parsing algorithms along with the data structures used, their limitations and strengths.
- Parser generators : how to automatically generate a parser from the cfg.

The details of the parsing algorithms shall be covered in subsequent lectures.

Principles of Top-Down Parsing

A top down parser creates a parse tree starting with the root, i. e., it starts a derivation from the start symbol of the grammar.

1. It could potentially replace any of the nonterminals in the current sentential form. However while tracing out a derivation , the goal is to produce the input string.
2. Since the input tokens are scanned from left to right , it will be desirable to construct a derivation that also produces terminals in the left to right fashion in the sentential forms.
3. A leftmost derivation matches the requirement exactly. A property of such a derivation is that there are only terminal symbols preceding the leftmost nonterminal in any leftmost sentential form.
4. The basic step in a top down parser is to find a candidate production rule (the one that could potentially produce a match for the input symbol under examination) in order to move from a left most sentential form to its succeeding left sentential form.
5. A top down parser uses a production rule in the obvious way - replace the lhs of the rule by one of its rhs.


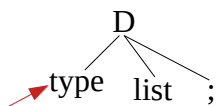
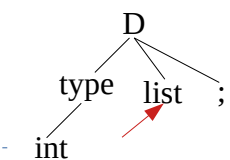
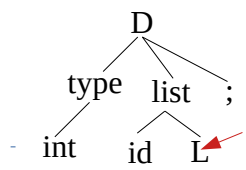
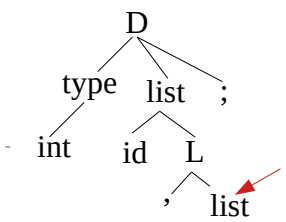
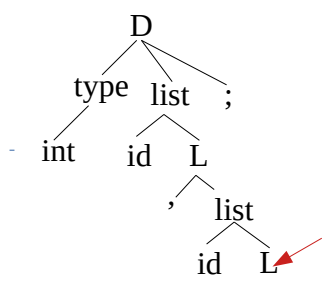
Example to Illustrate the Principles of Top-Down Parsing

Top-down parsing of the sentence “**int a, b;**” for the C declaration grammar, named as G1, given below.

| | |
|--------------------------------------|-------------------------------------|
| R1 : D \rightarrow type list ; | R2 : type \rightarrow int |
| R3 : type \rightarrow float | R4 : list \rightarrow id L |
| R5 : L \rightarrow , list | R6 : L \rightarrow ϵ |

The start symbol of the CFG is the nonterminal D. The rules have been numbered for ease of reference.

The processing of a top down parser over the given CFG and the input can be understood by tracing the moves of the parser. The current token in the input is highlighted and an arrow points to the leftmost terminal in the sentential form.

| Input | Rule used with Explanation | Incremental Generation of Parse Tree |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| int a , b ; | None | Initial Parse Tree  |
| int a , b ; | Leftmost nonterminal is D R1 : D → type list ; |  |
| int a , b ; | Leftmost nonterminal is type R2 : type → int int matches with the current token get next token; mark leftmost nonterminal |  |
| int a , b ; | Leftmost nonterminal is list R4 : list → id L a matches with current token id get next token; mark leftmost nonterminal |  |
| int a , b ; | Leftmost nonterminal is L R5 : L → , list , matches with current token , get next token; mark leftmost nonterminal |  |
| int a , b ; | Leftmost nonterminal is list R4 : list → id L b matches with current token id get next token; mark leftmost nonterminal |  |

| Input | Rule used with Explanation | Incremental Generation of Parse Tree |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| int a , b ; | Leftmost nonterminal is L $R6 : L \rightarrow \epsilon$ There are no leftmost nonterminal left in the parse tree The nexttoken ; matches with the ; of rule R1 | |

At this stage the input string is matched and exhausted. The parsing is successful and the yield of the parse tree is exactly the input string.

Q. How would Top Down parsing proceed if we had used the following equivalent grammar, say G2, instead ?

$R1 : D \rightarrow \text{type list ;}$
 $R2 : \text{type} \rightarrow \mathbf{int}$
 $R3 : \text{type} \rightarrow \mathbf{float}$
 $R4 : \text{list} \rightarrow \mathbf{id} , \text{list}$
 $R5 : \text{list} \rightarrow \mathbf{id}$

Processing CFG rules to generate some useful information about its nonterminals. Let us consider the earlier grammar.

$R1 : D \rightarrow \text{type list ;}$
 $R2 : \text{type} \rightarrow \mathbf{int}$
 $R3 : \text{type} \rightarrow \mathbf{float}$
 $R4 : \text{list} \rightarrow \mathbf{id} L$
 $R5 : L \rightarrow , \text{list}$
 $R6 : L \rightarrow \epsilon$

- A nonterminal including the start nonterminal can derive several sentential forms using the grammar rules. The nonterminals are {D, type, list, L}
- Consider the following derivations from D

$D \Rightarrow \text{type list ;} \Rightarrow \mathbf{int} \text{ list ;}$ $D \Rightarrow \text{type list ;} \Rightarrow \mathbf{float} \text{ list ;}$

While D can derive many more sentential forms, it can be seen that the first terminals that can appear in any sentential form (leftmost, rightmost or an arbitrary one) derivable from D are just two, namely {int, float}.

Similar information about the non-terminals of G1 can also be obtained. This information about a nonterminal, say A, is denoted by FIRST(A), and is very useful in parsing. For example, when the first token of a input, say “short”, is checked for syntactic validity with respect to G1, a top down parser will try to match “short” with FIRST(D) = {int, float} and since it fails will declare an error.

It appear at a first glance that to construct FIRST(A), for all A in the set of nonterminals, one would have to generate all the sentential forms. Fortunately that is not the case and these sets can be easily constructed using an algorithm that processes the rules iteratively.

Construction of FIRST() sets

The set of terminals that can ever appear in any sentential form derivable from a string α , $\alpha \in (N \cup T)^*$, is denoted by FIRST(α).

Formally, this is defined as $\text{FIRST}(\alpha) = \{ a \in T^* \mid \alpha \Rightarrow^* a \beta \text{ for some } \beta \}$.

It follows that $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

Some obvious facts about FIRST information are the following :

1. For a terminal a, $\text{FIRST}(a) = \{ a \}$
2. For a nonterminal A, if $A \rightarrow \epsilon$, then $\epsilon \in \text{FIRST}(A)$
3. For a rule $A \rightarrow X_1 X_2 \dots X_r$

$$\text{FIRST}(A) = \text{FIRST}(A) \cup_k (\text{FIRST}(X_k)); 1 \leq k \leq r; \epsilon \in X_i; 1 \leq i \leq k-1;$$

The expression above says that the contributions from the FIRST sets of X_2, X_3, \dots, X_{k-1} also need to be computed if all their preceding nonterminals can derive ϵ . For instance, the terminals in FIRST(X_5) are to be added to FIRST(A), only if, $\epsilon \in \text{FIRST}(X_1)$ and $\epsilon \in \text{FIRST}(X_2)$ and $\epsilon \in \text{FIRST}(X_3)$ and $\epsilon \in \text{FIRST}(X_4)$.

An Algorithm that computes FIRST sets of nonterminals for a general CFG follows.

Algorithm for FIRST sets

Input : Grammar $G = (N, T, P, S)$

Output : $FIRST(A)$, $A \in N$

Method :

begin algorithm

for $A \in N$ do $FIRST(A) = \Phi$;

 change := true;

 while change do

 { change := false;

 for $p \in P$ such that p is $A \rightarrow \alpha$ do

 { newFIRST(A) := $FIRST(A) \cup Y$; where Y is $FIRST(\alpha)$ from definition;

 if newFIRST(A) \neq FIRST(A) then

 { change := true ; FIRST(A) := newFIRST(A);}

 }

 }

end algorithm

Illustration of FIRST() Set Computation :

R1 : $D \rightarrow \text{type list}$; R2 : $\text{type} \rightarrow \text{int}$

R3 : $\text{type} \rightarrow \text{float}$ R4 : $\text{list} \rightarrow \text{id}$ L

R5 : $L \rightarrow , \text{list}$ R6 : $L \rightarrow \epsilon$

| Nonterminal | Initialization | Iteration 1 | Iteration 2 | Iteration 3 |
|-------------|----------------|-----------------|-----------------|-----------------|
| D | Φ | Φ | {int float} | {int float} |
| type | Φ | {int float} | {int float} | {int float} |
| list | Φ | {id} | {id} | {id} |
| L | Φ | { ϵ ,} | { ϵ ,} | { ϵ ,} |
| change | - | True | True | False |

The working of the algorithm can be explained as follows. The initialization step is obvious. In the first iteration, rule R1 is used for $FIRST(D) = FIRST(\text{type list} ;) = FIRST(\text{type}) = \Phi$

$FIRST(\text{type}) = FIRST(\text{int}) = \{\text{int}\}$ using R2 and using R3 $FIRST(\text{type}) = FIRST(\text{float}) = \{\text{int float}\}$.

Since $FIRST(\text{type})$ has changed from Φ to {int float}, change is set to True. Similarly in the second

iteration, $\text{FIRST}(D)$ changes due to which change is again set to True. All the $\text{FIRST}()$ sets in iteration 3 remain the same as they were at the end of iteration 2, which causes change to be assigned False and the algorithm terminates with the $\text{FIRST}()$ sets as given at the end of iteration 3.

Q. Construct a derivation that generates a sentential form justifying the presence of a terminal in the $\text{FIRST}(A)$ for any A .

Q. What is the worst complexity of the algorithm in terms of the quantities present in the grammar?

Another Useful Information From the CFG

Another information that is found to be very useful in several parsing methods (but not all) is also based on sentential forms that are generated by the grammar rules.

- Consider the following derivations from D using grammar G_1

| | |
|---------------------------------------------|--------------------------------------------|
| $R1 : D \rightarrow \text{type list ;}$ | $R2 : \text{type} \rightarrow \text{int}$ |
| $R3 : \text{type} \rightarrow \text{float}$ | $R4 : \text{list} \rightarrow \text{id L}$ |
| $R5 : L \rightarrow , \text{list}$ | $R6 : L \rightarrow \epsilon$ |

$D \Rightarrow \text{type list ;} \Rightarrow \text{int list ;} \quad D \Rightarrow \text{type list ;} \Rightarrow \text{type id L ;}$

$D \Rightarrow \text{type list ;} \Rightarrow \text{float list ;}$

$\text{list} \Rightarrow \text{id L} \Rightarrow \text{id , list} \Rightarrow \text{id , id L} \Rightarrow \dots$

- The information that is sought is about the terminals that appear immediately to the right of a nonterminal in any sentential form of G_1 . Such information is named as $\text{FOLLOW}(A)$ for a nonterminal A and gives the terminal symbols that immediately follow A in some sentential form of G_1 .
- For instance, using $R1$ it can be seen that $\text{FOLLOW}(\text{list})$ contains ‘;’ similarly from the derivation sequence, “ $D \Rightarrow \text{type list ;} \Rightarrow \text{type id L ;}$ ” we find that $\text{FOLLOW}(L)$ contains ‘;’. It is intended to construct $\text{FOLLOW}(A)$ for all A in N .

Rules for construction of $\text{FOLLOW}()$ sets are the following.

Rule 1 : For the start symbol of the grammar, $\$ \in \text{FOLLOW}(S)$. The rationale for this special rule is that it is essential to know when the entire input has been examined. A special marker, $\$$, is appended to be actual input. Since S is the start symbol, only the end of input marker can follow S .

Rule 2 : If $A \rightarrow \alpha B \beta$ is a grammar rule, then $\text{FOLLOW}(B) \cup = \{\text{FIRST}(\beta) - \epsilon\}$

Rule 3 : If either $A \rightarrow \alpha B$ is a rule or if $A \rightarrow \alpha B \beta$ is a rule and $\beta \Rightarrow^* \epsilon$ (that is, $\epsilon \in \text{FIRST}(\beta)$), then $\text{FOLLOW}(B) \cup = \text{FOLLOW}(A)$

The rule says that under the given conditions, whatever follows A also follows B.

- These rules can be applied to write an algorithm that constructs $\text{FOLLOW}(A)$ for all A in N iteratively similar to the algorithm for construction of FIRST sets.

G1 is reproduced below for ready reference.

$R1 : D \rightarrow \text{type list ;}$ $R2 : \text{type} \rightarrow \text{int}$
 $R3 : \text{type} \rightarrow \text{float}$ $R4 : \text{list} \rightarrow \text{id L}$
 $R5 : L \rightarrow \text{ , list}$ $R6 : L \rightarrow \epsilon$

- Rule 1 applied to R1 gives $\text{FOLLOW}(D) = \{\$ \}$.
Rule 2 applied to R1 gives $\text{FOLLOW}(\text{type}) \cup = \text{FIRST}(\text{list}) = \{\text{id}\}$;
 $\text{FOLLOW}(\text{list}) \cup = \text{FIRST}\{;\} = \{;\}$
- No Rule applies to R2 and R3.
- Rule 3 applied to R4 gives $\text{FOLLOW}(L) \cup = \text{FOLLOW}(\text{list}) = \{;\}$
- Rule 3 applied to R5 gives $\text{FOLLOW}(\text{list}) \cup = \text{FOLLOW}(L) = \{;\}$
- Processing of the $\text{FOLLOW}()$ sets iteratively shows that the information converges at the end of iteration 2.

| A in N | FIRST() | Initialization | Iteration 1 | Iteration 2 |
|---------------|------------------|----------------|-------------|-------------|
| D | {int float} | { \$ } | { \$ } | { \$ } |
| type | {int float} | Φ | {id} | {id} |
| list | {id} | Φ | { ; } | { ; } |
| L | { ϵ , } | Φ | { ; } | { ; } |
| change | | NA | True | False |

Q. Construct a derivation that generates a sentential form justifying the presence of a terminal in the $\text{FOLLOW}(A)$ for any A.

Q. Write an algorithm for construction of $\text{FOLLOW}()$ for all nonterminal symbols of a CFG.

Q. What is the worst complexity of your algorithm in terms of the quantities present in the grammar?

****** End of Document ******