



Compiler Design

Top Down Parsing

Amey Karkare

Department of Computer Science and Engineering

IIT Kanpur

karkare@iitk.ac.in

Top down Parsing

- Following grammar generates types of Pascal

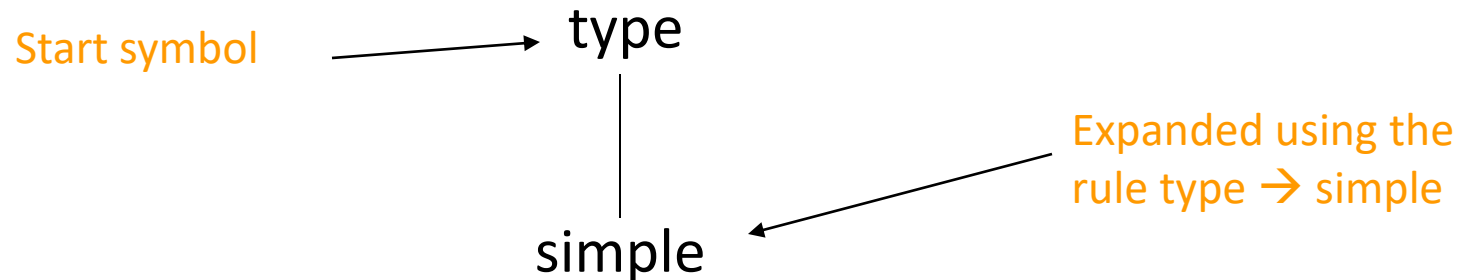
type \rightarrow simple
 | \uparrow id
 | array [simple] of type

simple \rightarrow integer
 | char
 | num dotdot num

Example ...

- Construction of a parse tree is done by starting the root labeled by a start symbol
- repeat following two steps
 - at a node labeled with non terminal A select one of the productions of A and construct children nodes
(Which production?)
 - find the next node at which subtree is Constructed
(Which node?)

- Parse
array [num dotdot num] of integer



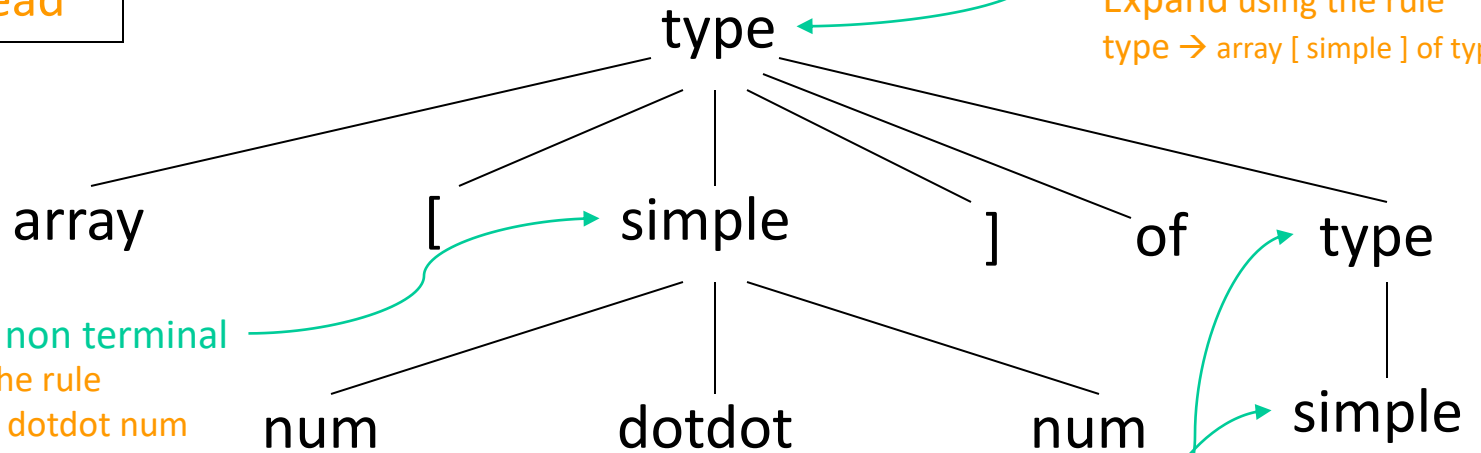
- Cannot proceed as non terminal “simple” never generates a string beginning with token “array”. Therefore, requires back-tracking.
- Back-tracking is not desirable, therefore, take help of a “look-ahead” token. The current token is treated as look-ahead token. (restricts the class of grammars)

array [num dotdot num] of integer

look-ahead

Start symbol

Expand using the rule
type \rightarrow array [simple] of type



Left most non terminal

Expand using the rule
Simple \rightarrow num dotdot num

all the tokens exhausted
Parsing completed

Left most non terminal

Expand using the rule
type \rightarrow simple

Left most non terminal

Expand using the rule
simple \rightarrow integer

Recursive descent parsing

First set:

Let there be a production

$$A \rightarrow \alpha$$

then $\text{First}(\alpha)$ is the set of tokens that appear as the first token in the strings generated from α

For example :

$\text{First}(\text{simple}) = \{\text{integer}, \text{char}, \text{num}\}$

$\text{First}(\text{num dotdot num}) = \{\text{num}\}$

Define a procedure for each non terminal

```
procedure type;
  if lookahead in {integer, char, num}
  then simple
  else if lookahead = ↑
    then begin match( ↑ );
           match(id)
        end
    else if lookahead = array
    then begin match(array);
           match([]);
           simple;
           match([]);
           match(of);
           type
        end
    else error;
```

```
procedure simple;  
  if lookahead = integer  
    then match(integer)  
  else if lookahead = char  
    then match(char)  
  else if lookahead = num  
    then begin match(num);  
              match(dotdot);  
              match(num)  
          end  
  else  
    error;
```

```
procedure match(t:token);  
  if lookahead = t  
    then lookahead = next token  
  else error;
```

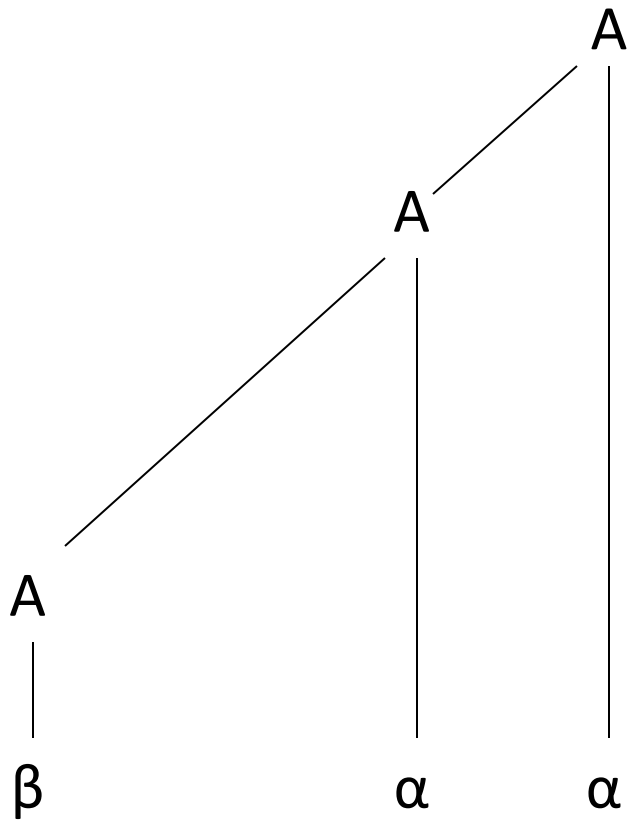

Left recursion

- A top down parser with production $A \rightarrow A \alpha$ may loop forever
- From the grammar $A \rightarrow A \alpha \mid \beta$ left recursion may be eliminated by transforming the grammar to

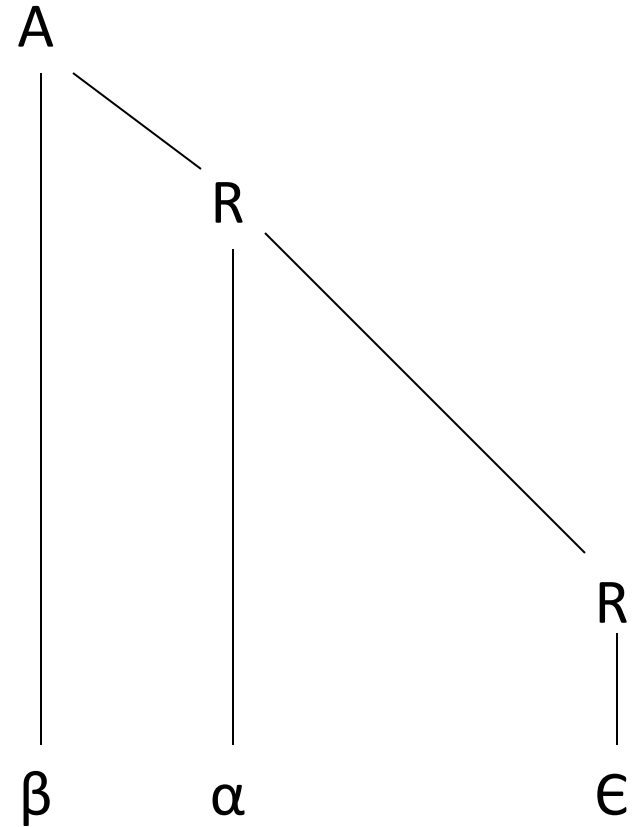
$$A \rightarrow \beta R$$

$$R \rightarrow \alpha R \mid \varepsilon$$

Parse tree corresponding
to a left recursive grammar



Parse tree corresponding
to the modified grammar



Both the trees generate string $\beta\alpha^*$

Example

- Consider grammar for arithmetic expressions

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

- After removal of left recursion the grammar becomes

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

Removal of left recursion

In general

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

transforms to

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

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

there is a left recursion because

$$S \rightarrow Aa \rightarrow Sda$$

- In such cases, left recursion is removed systematically
 - Starting from the first rule and replacing all the occurrences of the first non terminal symbol
 - Removing left recursion from the modified grammar

Removal of left recursion due to many productions ...

- After the first step (substitute S by its rhs in the rules) the grammar becomes

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- After the second step (removal of left recursion) the grammar becomes

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bdA' \mid A'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

Left factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol
defer the decision till we have seen enough input.

In general if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

defer decision by expanding A to $\alpha A'$

we can then expand A' to β_1 or β_2

- Therefore $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

transforms to

$$A \rightarrow \alpha A'$$

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

Dangling else problem again

Dangling else problem can be handled by left factoring

$\text{stmt} \rightarrow \text{if expr then stmt else stmt}$
 $\quad \quad | \text{if expr then stmt}$

can be transformed to

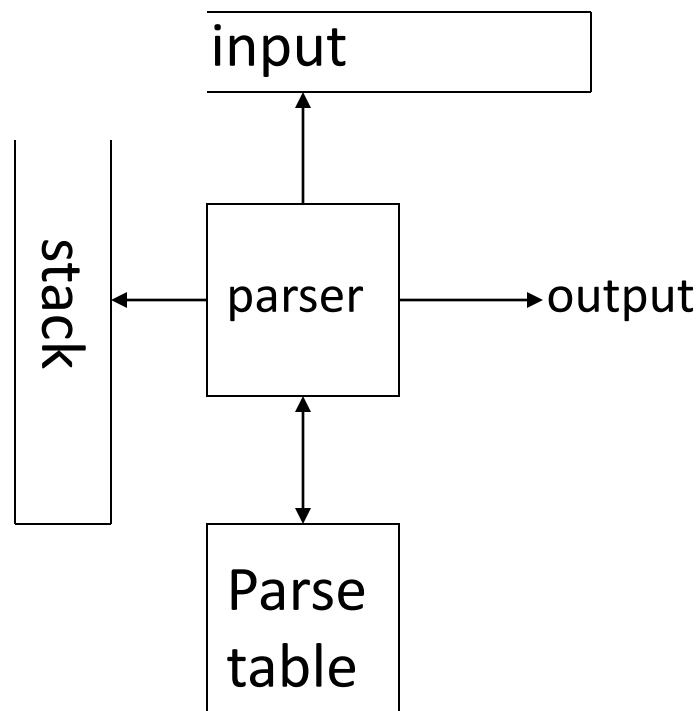
$\text{stmt} \rightarrow \text{if expr then stmt } S'$
 $S' \rightarrow \text{else stmt } | \epsilon$

Predictive parsers

- A non recursive top down parsing method
- Parser “predicts” which production to use
- It removes backtracking by fixing one production for every non-terminal and input token(s)
- Predictive parsers accept LL(k) languages
 - First L stands for left to right scan of input
 - Second L stands for leftmost derivation
 - k stands for number of lookahead token
- In practice LL(1) is used

Predictive parsing

- Predictive parser can be implemented by maintaining an external stack



Parse table is a two dimensional array $M[X,a]$ where “X” is a non terminal and “a” is a terminal of the grammar

Example

- Consider the grammar

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

Parse table for the grammar

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Blank entries are error states. For example
E cannot derive a string starting with '+'

Parsing algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol
- These two symbols determine the action to be taken by the parser
- Assume that '\$' is a special token that is at the bottom of the stack and terminates the input string

if $X = a = \$$ then halt

if $X = a \neq \$$ then pop(x) and ip++

if X is a non terminal

 then if $M[X,a] = \{X \rightarrow UVW\}$

 then begin pop(X); push(W,V,U)

 end

 else error

Example

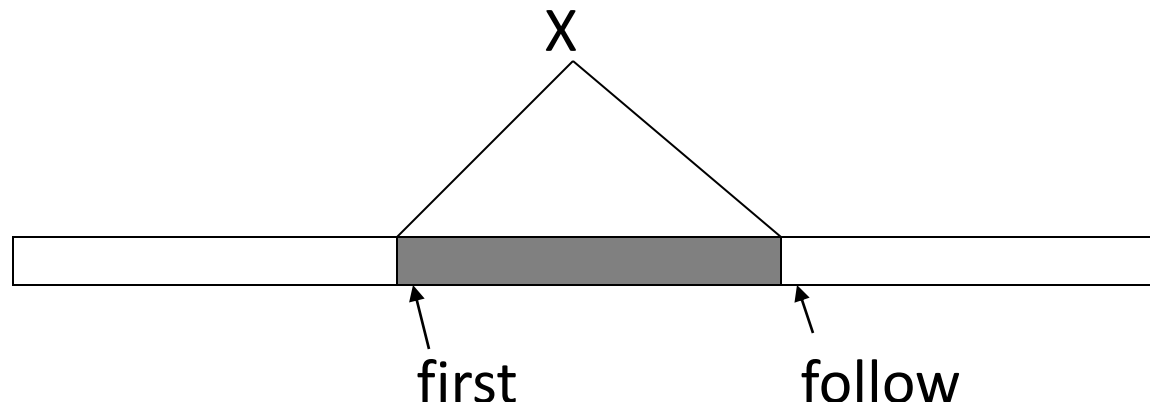
Stack	input	action
\$E	id + id * id \$	expand by $E \rightarrow TE'$
\$E'T	id + id * id \$	expand by $T \rightarrow FT'$
\$E'T'F	id + id * id \$	expand by $F \rightarrow id$
\$E'T'id	id + id * id \$	pop id and $ip++$
\$E'T'	+ id * id \$	expand by $T' \rightarrow \epsilon$
\$E'	+ id * id \$	expand by $E' \rightarrow +TE'$
\$E'T+	+ id * id \$	pop + and $ip++$
\$E'T	id * id \$	expand by $T \rightarrow FT'$

Example ...

Stack	input	action
\$E'T'F	id * id \$	expand by $F \rightarrow id$
\$E'T'id	id * id \$	pop id and ip++
\$E'T'	* id \$	expand by $T' \rightarrow *FT'$
\$E'T'F*	* id \$	pop * and ip++
\$E'T'F	id \$	expand by $F \rightarrow id$
\$E'T'id	id \$	pop id and ip++
\$E'T'	\$	expand by $T' \rightarrow \epsilon$
\$E'	\$	expand by $E' \rightarrow \epsilon$
\$	\$	halt

Constructing parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production
- $\text{First}(\alpha)$ for a string of terminals and non terminals α is
 - Set of symbols that might begin the fully expanded (made of only tokens) version of α
- $\text{Follow}(X)$ for a non terminal X is
 - set of symbols that might follow the derivation of X in the input stream



Compute first sets

- If X is a terminal symbol then $\text{First}(X) = \{X\}$
- If $X \rightarrow \epsilon$ is a production then ϵ is in $\text{First}(X)$
- If X is a non terminal
and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production
then
if for some i , a is in $\text{First}(Y_i)$
and ϵ is in all of $\text{First}(Y_j)$ (such that $j < i$)
then a is in $\text{First}(X)$
- If ϵ is in $\text{First}(Y_1) \dots \text{First}(Y_k)$ then ϵ is in $\text{First}(X)$

Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

$$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{ (, \text{id} \}$$

$$\text{First}(E') = \{ +, \epsilon \}$$

$$\text{First}(T') = \{ *, \epsilon \}$$

Compute follow sets

1. Place \$ in follow(S)
2. If there is a production $A \rightarrow \alpha B \beta$
then everything in first(β) (except ϵ) is in follow(B)
3. If there is a production $A \rightarrow \alpha B$
then everything in follow(A) is in follow(B)
4. If there is a production $A \rightarrow \alpha B \beta$
and First(β) contains ϵ
then everything in follow(A) is in follow(B)

Since follow sets are defined in terms of follow sets last two steps have to be repeated until follow sets converge

Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

$$\text{follow}(E) = \text{follow}(E') = \{ \$,) \}$$

$$\text{follow}(T) = \text{follow}(T') = \{ \$,), + \}$$

$$\text{follow}(F) = \{ \$,), +, * \}$$

Construction of parse table

- for each production $A \rightarrow \alpha$ do
 - for each terminal 'a' in $\text{first}(\alpha)$
 $M[A,a] = A \rightarrow \alpha$
 - If ϵ is in $\text{First}(\alpha)$
 $M[A,b] = A \rightarrow \alpha$
for each terminal b in $\text{follow}(A)$
 - If ϵ is in $\text{First}(\alpha)$ and $\$$ is in $\text{follow}(A)$
 $M[A,\$] = A \rightarrow \alpha$
- A grammar whose parse table has no multiple entries is called LL(1)

Practice

- Construct LL(1) parse table for the expression grammar
bexpr \rightarrow bexpr or bterm | bterm
bterm \rightarrow bterm and bfactor | bfactor
bfactor \rightarrow not bfactor | (bexpr) | true | false
- Steps to be followed
 - Remove left recursion
 - Compute first sets
 - Compute follow sets
 - Construct the parse table

Error handling

- Stop at the first error and print a message
 - Compiler writer friendly
 - But not user friendly
- Every reasonable compiler must recover from errors and identify as many errors as possible
- However, multiple error messages due to a single fault must be avoided
- Error recovery methods
 - Panic mode
 - Phrase level recovery
 - Error productions
 - Global correction

Panic mode

- Simplest and the most popular method
- Most tools provide for specifying panic mode recovery in the grammar
- When an error is detected
 - Discard tokens one at a time until a set of tokens is found whose role is clear
 - Skip to the next token that can be placed reliably in the parse tree

Panic mode ...

- Consider following code

begin

a = b + c;

x = p r ;

h = x < 0;

end;

- The second expression has syntax error
- Panic mode recovery for begin-end block
skip ahead to next ';' and try to parse the next expression
- It discards one expression and tries to continue parsing
- May fail if no further ';' is found

Phrase level recovery

- Make local correction to the input
- Works only in limited situations
 - A common programming error which is easily detected
 - For example insert a “;” after closing “}” of a class definition
- Does not work very well!

Error productions

- Add erroneous constructs as productions in the grammar
- Works only for most common mistakes which can be easily identified
- Essentially makes common errors as part of the grammar
- Complicates the grammar and does not work very well

Global corrections

- Considering the program as a whole find a correct “nearby” program
- Nearness may be measured using certain metric
- PL/C compiler implemented this scheme: anything could be compiled!
- It is complicated and not a very good idea!

Error Recovery in LL(1) parser

- Error occurs when a parse table entry $M[A,a]$ is empty
- Skip symbols in the input until a token in a selected set (synch) appears
- Place symbols in $\text{follow}(A)$ in synch set. Skip tokens until an element in $\text{follow}(A)$ is seen.
Pop(A) and continue parsing
- Add symbol in $\text{first}(A)$ in synch set. Then it may be possible to resume parsing according to A if a symbol in $\text{first}(A)$ appears in input.

Practice

- **Reading assignment:** Read about error recovery in LL(1) parsers
- **Practice Assignment:**
 - introduce synch symbols (using both follow and first sets) in the parse table created for the boolean expression grammar in the previous practice work
 - Parse “not (true and or false)” and show how error recovery works