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 → simple
| ↑ id
| array [simple] of type

simple → integer
| char
| num dotdot num

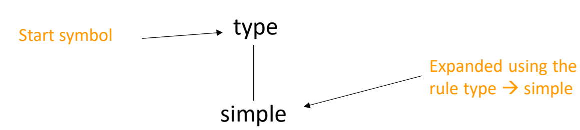
2

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?)

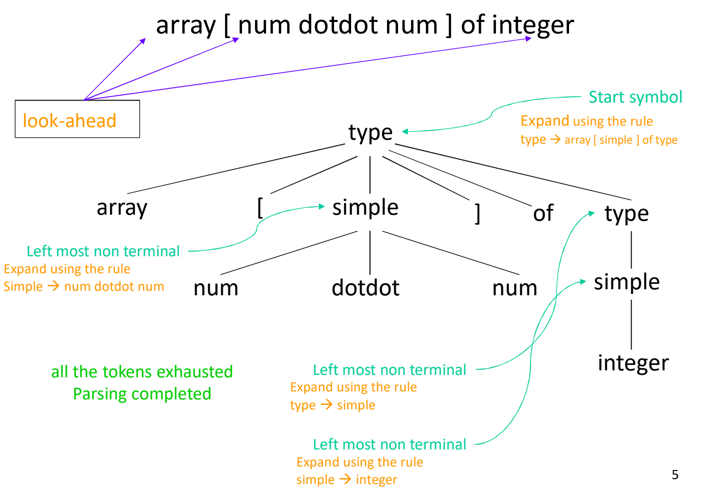
3

- 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)

4



5

Recursive descent parsing

First set:

Let there be a production

$A \rightarrow \alpha$

then First(α) is the set of tokens that appear as the first token in the strings generated from α

For example :

First(simple) = {integer, char, num}

First(num dotdot num) = {num}

6

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;
```

7

```
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;
```

8

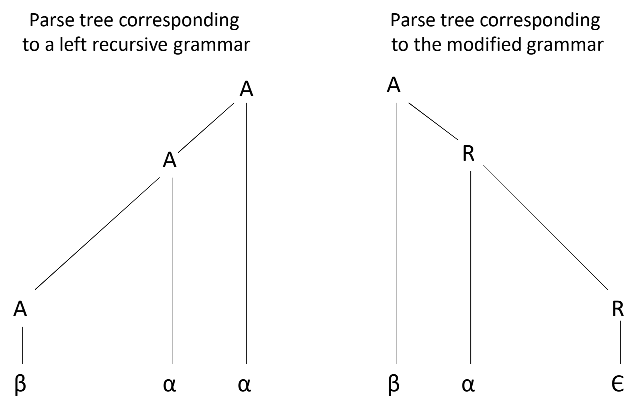
Left recursion

- 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

  $A \rightarrow \beta\ R$
  $R \rightarrow \alpha\ R \mid \varepsilon$

9



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

10

Example

- Consider grammar for arithmetic expressions

  E → E + T | T
  T → T * F | F
  F → (E) | id

- After removal of left recursion the grammar becomes

  E → T E’
  E’ → + T E’ | ϵ
  T → F T’
  T’ →* F T’ | ϵ
  F → (E) | id

11

Removal of left recursion

In general

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_m$
$\quad \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$

transforms to

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

12

Left recursion hidden due to many productions

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

  S → Aa | b
  A → Ac | Sd | ϵ

  there is a left recursion because

  S → Aa → 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 → Aa | b
  A → Ac | Aad | bd | ϵ

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

  S → Aa | b
  A → bdA' | A'
  A' → cA' | adA' | ϵ

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 → $\alpha\beta_1$ | $\alpha\beta_2$

  defer decision by expanding A to αA'

  we can then expand A' to β_1 or β_2

- Therefore A → $\alpha\ \beta_1$ | $\alpha\ \beta_2$

  transforms to

  A → αA'
  A' → β_1 | β_2

Dangling else problem again

Dangling else problem can be handled by left factoring

stmt → if expr then stmt else stmt
 | if expr then stmt

can be transformed to

stmt → if expr then stmt S'
S' → else stmt | ϵ

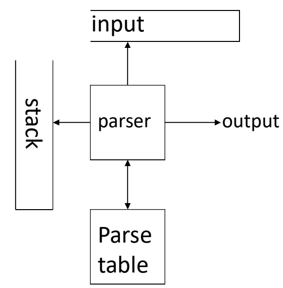
Predictive parsers

- 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

17

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

18

Example

- Consider the grammar

E → T E’
E' → +T E' | ϵ
T → F T'
T' → * F T' | ϵ
F → (E) | id

19

Parse table for the grammar

	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ϵ	E’→ϵ
T	T→FT’			T→FT’		
T’		T’→ϵ	T’→*FT’		T’→ϵ	T’→ϵ
F	F→id			F→(E)		

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

20

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 ≠ $ then pop(x) and ip++

if X is a non terminal
    then if M[X,a] = {X → UVW}
        then begin pop(X); push(W,V,U)
            end
        else error

Example

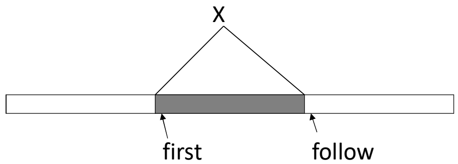
Stack	input	action
$E	id + id * id $	expand by E→TE'
$E'T	id + id * id $	expand by T→FT'
$E'T'F	id + id * id $	expand by F→id
$E'T'id	id + id * id $	pop id and ip++
$E'T'	+ id * id $	expand by T'→Є
$E'	+ id * id $	expand by E'→+TE'
$E'T+	+ id * id $	pop + and ip++
$E'T	id * id $	expand by T→FT'

Example …

Stack	input	action
$E'T'F	id * id $	expand by F→id
$E'T'id	id * id $	pop id and ip++
$E'T'	* id $	expand by T'→*FT'
$E'T'F*	* id $	pop * and ip++
$E'T'F	id $	expand by F→id
$E'T'id	id $	pop id and ip++
$E'T'	$	expand by T'→Є
$E'	$	expand by E'→Є
$	$	halt

Constructing parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production
- First(α) 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
- 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 First(X) = {X}
- If X → ∈ is a production then ∈ is in First(X)
- If X is a non terminal
  and X → $Y_l Y_2 \ldots Y_k$ is a production
  then
  if for some i, a is in First(Y_i)
  and ∈ is in all of First(Y_j) (such that j<i)
  then a is in First(X)
- If ∈ is in First (Y_1) … First(Y_k) then ∈ is in First(X)

Example

- For the expression grammar
  E → T E'
  E' → +T E' | ∈
  T → F T'
  T' → * F T' | ∈
  F → (E) | id

  First(E) = First(T) = First(F) = { (, id }
  First(E') = {+, ∈}
  First(T') = { *, ∈}

Compute follow sets

1. Place $ in follow(S)
2. If there is a production A → αBβ
   then everything in first(β) (except ε) is in follow(B)
3. If there is a production A → αB
   then everything in follow(A) is in follow(B)
4. If there is a production A → αBβ
   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 → T E'
  E' → + T E' | ∈
  T → F T'
  T' → * F T' | ∈
  F → (E) | id

  follow(E) = follow(E') = { $,) }
  follow(T) = follow(T') = { $,), + }
  follow(F) = { $,), +, *}

Construction of parse table

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

29

Practice

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

30

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

31

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

32

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

33

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!

34

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

35

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!

36

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 follow(A) in synch set. Skip tokens until an element in follow(A) is seen.

  Pop(A) and continue parsing
- Add symbol in first(A) in synch set. Then it may be possible to resume parsing according to A if a symbol in first(A) appears in input.

37

Practice

- Reading assignment: Read about error recovery in LL(1) parsers
- 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

38