Name Manas Vinod Darpelli

Roll 27

Div A

Tybsc(cs)

Role of parser

Context free grammars

Top down parsing

Bottom up parsing

Parser generators

The role of parser

Lexical

Analyzer Parser Source

program

token

getNext

Token

Symbol

table

Parse tree Rest of

Front

End

Intermediate representation

Uses of grammars

Error handling

Common programming errors

Lexical errors

Syntactic errors

Semantic errors

Lexical errors

Error handler goals

Report the presence of errors clearly and accurately

Recover from each error quickly enough to detect subsequent errors

Add minimal overhead to the processing of correct progrms

Error-recover strategies

Panic mode recovery

Discard input symbol one at a time until one of esignated set of synchronization tokens is found

Phrase level recovery

Replacing a prefix of remaining input by some string
hat allows the parser to continue
Error productions
Augment the grammar with productions that generate
the erroneous constructs
Global correction
Choosing minimal sequence of changes to obtain a
globally least-cost correction
Context free grammars
Terminals
Nonterminals
Start symbol
productions
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term factor
term -> factor
factor -> (expression)
factor -> id

Derivations
Productions are treated as rewriting rules to generate

a

string

Rightmost and leftmost derivations

E -> E + E | E * E | -E | (E) | id

Derivations for –(id+id)

E => -E => -(E) => -(E+E) => -(id+E)=>-(id+id)

Parse trees

-(id+id) E => -E => -(E) => -(E+E) => -(id+E)=>-(id+id)

Ambiguity

For some strings there exist more than one parse tree

Or more than one leftmost derivation

Or more than one rightmost derivation

Example: id+id*id

Elimination of ambiguity

Elimination of ambiguity (cont.)

Idea:

A statement appearing between a then and an else

must be matched

Elimination of left recursion

A grammar is left recursive if it has a non-terminal A

such that there is a derivationA=> Aα Top down parsing methods cant

handle left recursive grammars

A simple rule for direct leftrecursion elimination: For a rule like: A ->

A

Left recursion elimination (cont.)

There are cases like following

S -> Aa / b

A -> Ac / Sd / ε Left recursion elimination algorithm:

Arrange the nonterminals in some order A1,A2,...,An.

For (each i from 1 to n) {

For (each j from 1 to i-1) {

Replace each production of the form

are all current Aj productions

} Eliminate left recursion among the Ai-productions

}

Left factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.

Consider following grammar:

Stmt -> if expr then stmt else stmt

/ if expr then stmt

On seeing input if it is not clear for the parser which production to useWe can easily perform left factoring:

Left factoring

(cont.)

Algorithm

For each non-terminal A, find the longest prefix α

common to two or more of its alternatives.

Example: S -> I E t S | i E t S e S | a

E -> b

Introduction

A Top-down parser tries to create a parse tree from the

root towards the leafs scanning input from left to right

It can be also viewed as finding a leftmost derivation

for an input string

Recursive descent parsing

Consists of a set of procedures, one for each

nonterminal

Execution begins with the procedure for start symbol

A typical procedure for a non-terminal

void A() {

choose an A-production, A->X1X2..Xk

for (i=1 to k) {

if (Xi is a nonterminal

call procedure Xi();

else if (Xi equals the current input symbol a)

advance the input to the next

symbol;

else /* an error has occurred */

} }

Recursive descent parsing (cont)

General recursive descent may require backtracking

The previous code needs to be modified to allow backtracking

In general form it cant choose an A-production easily.

So we need to try all alternatives

If one failed the input pointer needs to be reset and another alternative should be tried

Recursive descent parsers cant be used for left recursive gramma

Example

S->cAd

A->ab | a Input: cad

S c A d S c A d a b S c A d a

First and Follow

First() is set of terminals that begins strings derived from

and First are disjoint sets then we can select appropriate A-production by looking at the next input

Follow, for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential

form

If we have S => for some α and then a is in

Follow(A)

If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:

1. If X is a terminal then First(X) = {X}.
2. If X is a nonterminal and X->Y1Y2...Yk is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ε is in all of First(Y1),...,First(Yi-1) that is Y1...Yi-1 => ε. if ε is in First(Yj) for j=1,...,k then add ε to First(X).
3. If X-> ε is a production then add ε to First(X)

Example

Computing follow

To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow

set:

1. Place $ in Follow(S) where S is the start symbol

2. If there is a production A->

then everything in

First except ε is in Follow(B).

3. If there is a production A->B or a production

where First contains then everything

in Foll(A) is in Follow(B)

Example!

LL(1) Grammars

Predictive parsers are those recursive descent parsers needing no backtracking

Grammars for which we can create predictive parsers are called LL(1)

The first L means scanning input from left to right

The second L means leftmost derivation

And 1 stands for using one input symbol for lookahead

A grammar G is LL(1) if and only if whenever A-> α/β are two distinct productions of G, the following conditions hold:

For no terminal a do α and β both derive strings beginning with a

At most one of α or β can derive empty string

* If α=> ε then β does not derive any string beginning with a terminal in Follow(A).

Construction of

predictive

parsing table

For each production in grammar do the

following:

1. For each terminal a in First() add A-> in M[A,a]

2. If ε is in First(α), then for each terminal b in

Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and

$ is in Follow(A), add A-> ε to M[A,$] as well

If after performing the above, there is no production

in M[A,a] then set M[A,a] to error

Example

E -> TE'

E' -> +TE' |

T -> FT'

T' -> *FT' |

F -> (E) | id

FTE

E'

T'

First

Follow

{(,id}

{(,id}

{(,id}

{+,ε}

{*,ε}

{+, *,),$}

{+,),$}

{+,),$}

{), $}

{), $}

E

E'

T

T'

F

Non -

terminal

Input Symbol

id + * ( )$

E -> TE' E -> TE'

E' -> +TE' E' -> ε E' -> ε

T -> FT' T -> FT'

T' -> ε T' -> *FT' T' -> ε T' -> ε

F -> id F ->

(E)

Another example

S -> iEtSS" / a

S" -> eS / ε

E -> b

S

S"

E

Non -

terminal

Input Symbol

a b e i t $

S -> a S -> iEtSS"

S" -> ε

S" -> eS

S" -> ε

E -> b

Non-recursive predicting parsing

a + b

$

Predictive

parsing

program

output

Parsing

Table

M

stack XYZ$

Predictive parsing algorithm

Set ip point to the first symbol of w;

Set X to the top stack symbol;

While (X<>$) { /* stack is not empty */

if (X is a) pop the stack and advance ip;

else if (X is a terminal) error();

else if (M[X,a] is an error entry) error();

else if (M[X,a] = X->Y1Y2..Yk) {

output the production X->Y1Y2..Yk;

pop the stack;

push Yk,...,Y2,Y1 on to the stack with Y1 on top;

}

set X to the top stack

symbol;

}

Example

 id+id*id$

Matched Stack Input Action

E$ id+id*id$

Error recovery in predictive parsing

 Panic mode

 Place all symbols in Follow(A) into synchronization set for nonterminal A: skip tokens until an element of Follow(A) is seen and pop A from stack.

 Add to the synchronization set of lower level construct the symbols that begin higher level constructs

 Add symbols in First(A) to the synchronization set of nonterminal A If a nonterminal can generate the empty string then the production deriving can be used as a default

 If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was insterted

Example E

E'

T

T'

F

Non

-

terminal

Input Symbol

id + * ( )$

E -> TE' E -> TE'

E' -> +TE' E' -> Ɛ E' ->

T -> FT' T -> FT'

T' -> Ɛ T' -> *FT' T' -> T' ->

F -> id F -> (E)

synch synch

synch synch synch

synch synch synch synch

Stack Input Action

E$ id*+id$ Error, Skip)

E$ id*+id$ id is in First(E)

TE'$ id*+id$

FT'E'$ id*+id$

idT'E'$ id*+id$

T'E'$ *+id$

*FT'E'$ *+id$

FT'E'$ +id$ Error, M[F,+]=synch

T'E'$ +id$ F has been poped

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the

root

(the top)

 Example: id*id

E -> E + T | T

T -> T * F | F

F -> (E) | id id

id*id F * id T * id

id

F

T * F

id

F id T * F

id

F id

F

T * F

id

F id

FE

Shift-reduce parser

The general idea is to shift some symbols of input to

the stack until a reduction can be applied

At each reduction step, a specific substring matching

the body of a production is replaced by

the

nonterminal at the head of the production

The key decisions during bottom-up parsing are about

when to reduce and about what production to apply

A reduction is a reverse of a step in a derivation

The goal of a bottom-up parser is to construct a

derivation in reverse:

E=>T=>T*F=>T*id=>F*id=>id*id

Handle pruning

A Handle is a substring that matches the body of a

production and whose reduction represents one step

along the reverse of a rightmost derivation

Right sentential form Handle Reducing production

id*id id F->id

F*id F

id

T->F

T*id F->id

T*F T*F E->T*F

Shift reduce parsing

A stack is used to hold grammar symbols

Handle always appear on top of the stack

Initial configuration:

Stack

Input

$ w$

Acceptance configuration

Stack Input

$S $

Shift reduce parsing (cont.)

Basic operations:

Shift

Reduce

Accept

Error

Example: id*id

Stack Input Action

$

$id

id*id$ shift

*id$ reduce by F->id

$F *id$ reduce by T->F

$T *id$ shift

$T* id$ shift

$T*id $ reduce by F->id

$T*F $ reduce by T->T*F

$T $ reduce by E->T

$E $

accept

Handle will appear on top of

the stack

Conflicts during shit reduce

parsing

Two kind of conflicts

Shift/reduce conflict

Reduce/reduce conflict

Example:

Stack Input

... if expr then stmt else ....$

Reduce/reduce conflict

stmt -> id(parameter_list)

stmt -> expr:=expr

parameter_list->parameter_list, parameter

parameter_list->parameter

parameter->id

expr->id(expr_list)

expr->id

expr_list->expr_list, expr

expr_list->expr Stack Input

... id(id id) ....$

LR

Parsing

The most prevalent type of bottom-up parsers

LR(k), mostly interested on parsers with k<=1

Why LR parsers?

Table driven

Can be constructed to recognize all programming language constructs

Most general non-backtracking shift-reduce parsing method

Can detect a syntactic error as soon as it is possible to do so

Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

States represent set of items

An LR(0) item of G is a production of G with the dot at some position of the body:

For A->XYZ we have following items

A->.XYZ

A->X.YZ

A->XY.Z

A->XYZ.

In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.

- What about A->X.YZ?

Constructing canonical

LR(0)

item sets

Augmented grammar:

G with addition of a production: S"->S

Closure of item sets:

If I is a set of items, closure(I) is a set of items constructed from I by the following rules:

Add every item in I to closure(I)

If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).

Example: E'->E

E -> E + T / T

T -> T * F / F

F -> (E) / id

I0=closure({[E'->.E]}

E'->.E

E->.E+T

E->.T

T->.T*F

T->.F

F->.(E)

F->.id

Constructing canonical LR(0)

item sets

(cont.)

Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> αX. β] where [A-> α.Xβ] is in