Lecture 8

Syntax Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

February 5, 2025

Take aways from the last class

- Predictive Parser

Take aways from the last class

- Predictive Parser
- Parsing algorithm

Take aways from the last class

- Predictive Parser
- Parsing algorithm
- Computing *first* for grammar symbols.

Compute follow sets

Compute follow sets

- Place $ in $follow(S)$

Compute follow sets

- Place \$ in $follow(S)$
- If there is a production $A \rightarrow \alpha B \beta$ then everything in $first(\beta)$ (except ϵ) is in $follow(B)$

Compute follow sets

- Place \$ in $follow(S)$
- If there is a production $A \rightarrow \alpha B \beta$ then everything in $first(\beta)$ (except ϵ) is in $follow(B)$
- If there is a production $A \rightarrow \alpha B$ then everything in $follow(A)$ is in $follow(B)$

Compute follow sets

- Place \$ in *follow*(S)
- If there is a production $A \rightarrow \alpha B \beta$ then everything in *first*(β) (except ϵ) is in *follow*(B)
- If there is a production $A \rightarrow \alpha B$ then everything in *follow*(A) is in *follow*(B)
- If there is a production $A \rightarrow \alpha B \beta$ and *First*(β) contains ϵ then everything in *follow*(A) is in *follow*(B)

Example

- For the expression grammar
  $E \rightarrow TE'$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$

Example

- For the expression grammar
  $E \rightarrow TE'$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$
- $follow(E) = follow(E') = \$,)$

Example

- For the expression grammar
  $E \rightarrow TE'$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$
- $follow(E) = follow(E') = \$,)$
- $follow(T) = follow(T') = \$,), +$

Example

- For the expression grammar
  $E \rightarrow TE'$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$
- $follow(E) = follow(E') = \$,)$
- $follow(T) = follow(T') = \$,), +$
- $follow(F) = \$,), +, *$

Example

- For the expression grammar
  $E \rightarrow TE'$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$
- $follow(E) = follow(E') = \$,)$
- $follow(T) = follow(T') = \$,), +$
- $follow(F) = \$,), +, *$

Construction of parse table

Construction of parse table

- for each production $A \rightarrow \alpha$ do

Construction of parse table

- for each production $A \rightarrow \alpha$ do
  - for each terminal 'a' in $first(\alpha)$
    $M[A, a] = A \rightarrow \alpha$

Construction of parse table

- for each production $A \rightarrow \alpha$ do
  - for each terminal 'a' in $first(\alpha)$
    $M[A, a] = A \rightarrow \alpha$
  - If ϵ is in $First(\alpha)$
    $M[A, b] = A \rightarrow \alpha$
    for each terminal b in $follow(A)$

Construction of parse table

- for each production $A \rightarrow \alpha$ do
  - for each terminal 'a' in $first(\alpha)$
    $M[A, a] = A \rightarrow \alpha$
  - If ϵ is in $First(\alpha)$
    $M[A, b] = A \rightarrow \alpha$
    for each terminal b in $follow(A)$
  - If ϵ is in $First(a)$ and \$ is in $follow(A)$
    $M[A, \$] = A \rightarrow \alpha$

Construction of parse table

- for each production $A \rightarrow \alpha$ do
  - for each terminal 'a' in $first(\alpha)$
    $M[A, a] = A \rightarrow \alpha$
  - If ϵ is in $First(\alpha)$
    $M[A, b] = A \rightarrow \alpha$
    for each terminal b in $follow(A)$
  - If ϵ is in $First(a)$ and \$ is in $follow(A)$
    $M[A, \$] = A \rightarrow \alpha$

Here, no "accept" as in bottom up parsers.
Accept will be there when x = a = \$.

- A grammar whose parse table has no multiple entries is called $LL(1)$

Error handling

Error handling

- Stop at the first error and print a message

Error handling

- Stop at the first error and print a message
  - Compiler writer friendly

Error handling

- Stop at the first error and print a message
  - Compiler writer friendly
  - But not user friendly

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

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 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

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 model

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 model
  - Phase level recovery

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 model
  - Phase level recovery
  - Error production

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 model
  - Phase level recovery
  - Error production
  - Global correction

4 error recovery methods in top down parsing.

Panic Model

- Simplest and the most popular method

Panic Model

- Simplest and the most popular method
- Most tools provide for specifying panic mode recovery in the grammar

Panic Model

- Simplest and the most popular method
- Most tools provide for specifying panic mode recovery in the grammar
- When an error is detected

Panic Model

- 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

Panic Model

- 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 Model

- 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
- Consider following code

  $a = b + c;$

  $x = p \quad r;$

  $h = x < 0;$

Panic Model

- 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
- Consider following code

  $a = b + c;$

  $x = p \quad r;$

  $h = x < 0;$
- Panic mode recovery for block skip ahead to next ';' and try to parse the next expression

Panic Model

- 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
- Consider following code

  $a = b + c;$

  $x = p \quad r;$

  $h = x < 0;$
- Panic mode recovery for block skip ahead to next ';' and try to parse the next expression
- It discards one expression and tries to continue parsing

Panic Model

some special set of tokens are found.

- 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
- Consider following code

  $a = b + c;$

  $x = p \quad r;$

  $h = x < 0;$
- Panic mode recovery for 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

Panic Model

- 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
- Consider following code

  $a = b + c;$

  $x = p \quad r;$

  $h = x < 0;$
- Panic mode recovery for 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

Phase Level Recovery

Phase Level Recovery

- Make local correction to the input

Phase Level Recovery

- Make local correction to the input
- Works only in limited situations

Phase Level Recovery

- Make local correction to the input
- Works only in limited situations
  - A common programming error which is easily detected

Phase 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

Phase 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 Production

Error Production

- Add erroneous constructs as productions in the grammar

Error Production

- Add erroneous constructs as productions in the grammar
- Works only for most common mistakes which can be easily identified

Error Production

- 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

Error Production

- 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 Correction

Global Correction

- Considering the program as a whole find a correct “nearby” program

Global Correction

- Considering the program as a whole find a correct “nearby” program
- Nearness may be measured using certain metric

Global Correction

- 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!

Global Correction

- 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 Recovery in LL(1) parser

- Error occurs when a parse table entry $M[A, a]$ is empty

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

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

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.

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.

Bottom up parsing

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar
  $S \rightarrow aABe$
  $A \rightarrow Abc|b$
  $B \rightarrow d$

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

  Reduction of the string

  a b b c d e

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

  Reduction of the string

  a b b c d e

  a A b c d e

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

  Reduction of the string

  a b b c d e

  a A b c d e

  a A d e

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

  Reduction of the string

  a b b c d e

  a A b c d e

  a A d e

  a A B e

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

  Reduction of the string

  a b b c d e

  a A b c d e

  a A d e

  a A B e

  S

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

Reduction of the string

a b b c d e

a A b c d e

a A d e

a A B e

S

Rightmost derivation

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

Reduction of the string

a b b c d e

a A b c d e

a A d e

a A B e

S

Rightmost derivation

$S \rightarrow aABe$

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

Reduction of the string

a b b c d e

a A b c d e

a A d e

a A B e

S

Rightmost derivation

$S \rightarrow aABe$

$S \rightarrow aAde$

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

Reduction of the string

a b b c d e

a A b c d e

a A d e

a A B e

S

Rightmost derivation

$S \rightarrow aABe$

$S \rightarrow aAde$

$S \rightarrow aAbcde$

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root
- Reduce a string w of input to start symbol of grammar
- Consider the following grammar

  $S \rightarrow aABe$

  $A \rightarrow Abc|b$

  $B \rightarrow d$

Reduction of the string

a b b c d e

a A b c d e

a A d e

a A B e

S

Rightmost derivation

$S \rightarrow aABe$

$S \rightarrow aAde$

$S \rightarrow aAbcde$

$S \rightarrow abbcde$

bottom up parsing will give reverse of rightmost derivation.

Shift reduce parsing

Shift reduce parsing

- Split string being parsed into two parts

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is Bottom up parsing $.w$

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is Bottom up parsing $.w$
- Bottom up parsing has two actions

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is Bottom up parsing *.w*
- Bottom up parsing has two actions
- Shift: move terminal symbol from right string to left string if string before shift is then string after shift is $\alpha.pqr$ $\alpha p.qr$

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is Bottom up parsing *.w*
- Bottom up parsing has two actions
- Shift: move terminal symbol from right string to left string if string before shift is then string after shift is $\alpha.pqr$ $\alpha p.qr$
- Reduce: immediately on the left of "." identify a string same as RHS of a production and replace it by LHS if string before reduce action is $\alpha\beta.pqr$ and $A \rightarrow \beta$ is a production then string after reduction is $\alpha A.pqr$

note that in left part, the string to reduce will be a suffix always in bottom up parsing.