Lecture 13

Semantics Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

February 18, 2025

Take aways from the last class

- LR(1) parse table

Take aways from the last class

- LR(1) parse table
- LALR Parse Table

Take aways from the last class

- LR(1) parse table
- LALR Parse Table
- Error Recovery

Take aways from the last class

- LR(1) parse table
- LALR Parse Table
- Error Recovery
- Parser Generator

Semantic Analysis

- Check semantics

Semantic Analysis

- Check semantics
- Error reporting

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate
- Overloaded operators

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate
- Overloaded operators
- Static checking

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate
- Overloaded operators
- Static checking
    - Type checking

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate
- Overloaded operators
- Static checking
  - Type checking
  - Control flow checking

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate
- Overloaded operators
- Static checking
  - Type checking
  - Control flow checking
  - Uniqueness checking

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate
- Overloaded operators
- Static checking
    - Type checking
    - Control flow checking
    - Uniqueness checking
    - Name checks

Beyond syntax analysis

- Parser cannot catch all the program errors

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
    - Whether an identifier has been declared before use

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $w\alpha w|$ w $\in \Sigma$*

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $w\alpha w|$ w $\in \Sigma$*
  - This language is not deterministic context free

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $w\alpha w|$ w $\in \Sigma$*
  - This language is not deterministic context free
  - `string x; int y;`
    `y=x+3`
    the use of x is a type error

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $w\alpha w|$ w $\in \Sigma^*$
  - This language is not deterministic context free
  - `string x; int y;`
    `y=x+3`
    the use of x is a type error
  - An identifier may refer to different variables in different parts of the program

Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $w\alpha w|$ w $\in \Sigma^*$
  - This language is not deterministic context free
  - `string x; int y;`
    `y=x+3`
    the use of x is a type error
  - An identifier may refer to different variables in different parts of the program
  - An identifier may be usable in one part of the program but not another

Compiler needs to know

- Whether a variable has been declared?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like $A[i, j, k]$ is consistent with the declaration?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like $A[i, j, k]$ is consistent with the declaration?
- How many arguments does a function take?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like $A[i, j, k]$ is consistent with the declaration?
- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like $A[i, j, k]$ is consistent with the declaration?
- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?
- If an operator/function is overloaded, which function is being invoked?

Compiler needs to know

- Whether a variable has been declared?
- Are there variables that have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like $A[i, j, k]$ is consistent with the declaration?
- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?
- If an operator/function is overloaded, which function is being invoked?
- Methods in a class are not multiply defined

How to answer these questions?

- These issues are part of semantic analysis phase

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
    - Context sensitive grammars

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
    - Context sensitive grammars
- Use ad-hoc techniques

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
  - Context sensitive grammars
- Use ad-hoc techniques
  - Symbol table

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
  - Context sensitive grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
  - Context sensitive grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
  - Context sensitive grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between
  - Use attributes

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
  - Context sensitive grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between
  - Use attributes
  - Do analysis along with parsing

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases
- Use formal methods
  - Context sensitive grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between
  - Use attributes
  - Do analysis along with parsing
  - Use code for attribute value computation

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
    - Syntax directed definition
        - High level specifications
        - Hides implementation details
        - Explicit order of evaluation is not specified

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details
    - Explicit order of evaluation is not specified
  - Translation schemes

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details
    - Explicit order of evaluation is not specified
  - Translation schemes
    - Indicate order in which semantic rules are to be evaluated

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details
    - Explicit order of evaluation is not specified
  - Translation schemes
    - Indicate order in which semantic rules are to be evaluated
    - Allow some implementation details to be shown

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details
    - Explicit order of evaluation is not specified
  - Translation schemes
    - Indicate order in which semantic rules are to be evaluated
    - Allow some implementation details to be shown
- Conceptually both:

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
    - Syntax directed definition
        - High level specifications
        - Hides implementation details
        - Explicit order of evaluation is not specified
    - Translation schemes
        - Indicate order in which semantic rules are to be evaluated
        - Allow some implementation details to be shown
- Conceptually both:
    - Parse input token stream

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
    - Syntax directed definition
        - High level specifications
        - Hides implementation details
        - Explicit order of evaluation is not specified
    - Translation schemes
        - Indicate order in which semantic rules are to be evaluated
        - Allow some implementation details to be shown
- Conceptually both:
    - Parse input token stream
    - Build parse tree

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details
    - Explicit order of evaluation is not specified
  - Translation schemes
    - Indicate order in which semantic rules are to be evaluated
    - Allow some implementation details to be shown
- Conceptually both:
  - Parse input token stream
  - Build parse tree
  - Traverse the parse tree to evaluate the semantic rules at the parse tree nodes

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - High level specifications
    - Hides implementation details
    - Explicit order of evaluation is not specified
  - Translation schemes
    - Indicate order in which semantic rules are to be evaluated
    - Allow some implementation details to be shown
- Conceptually both:
  - Parse input token stream
  - Build parse tree
  - Traverse the parse tree to evaluate the semantic rules at the parse tree nodes
- It may store information in symbol table

Attributes

- Attributes fall into two classes:

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes
- Each grammar production $A \to \alpha$ has associated with it a set of semantic rules of the form

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes
- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form
  $b = f(c_1, c_2, \cdots, c_k)$
  - b is a synthesized attribute of A, OR

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes
- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form
  $b = f(c_1, c_2, \cdots, c_k)$
  - b is a synthesized attribute of A, OR
  - b is an inherited attribute of one of the grammar symbols on the right

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes
- Each grammar production $A \to \alpha$ has associated with it a set of semantic rules of the form
  $b = f(c_1, c_2, \cdots, c_k)$
  - b is a synthesized attribute of A, OR
  - b is an inherited attribute of one of the grammar symbols on the right
- The synthesized attribute of Node `N` can be defined using inherited attributes of Node `N`

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes
- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form
  $b = f(c_1, c_2, \cdots, c_k)$
  - b is a synthesized attribute of A, OR
  - b is an inherited attribute of one of the grammar symbols on the right
- The synthesized attribute of Node `N` can be defined using inherited attributes of Node `N`
- Inherited attribute of Node `N` can not be defined using attribute of child of Node `N`

Attributes

- Attributes fall into two classes:
  - Synthesized: value of a synthesized attribute is computed from the values of its children nodes and at itself
  - Inherited: value of an inherited attribute is computed from the sibling, itself and its parent nodes
- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form
  $b = f(c_1, c_2, \cdots, c_k)$
  - b is a synthesized attribute of A, OR
  - b is an inherited attribute of one of the grammar symbols on the right
- The synthesized attribute of Node `N` can be defined using inherited attributes of Node `N`
- Inherited attribute of Node `N` can not be defined using attribute of child of Node `N`
- Terminal can have only synthesized attributes (calculated from lexical phase). No SDD rules for computing attributes of terminal

Example

- Consider a grammar for signed binary numbers
  $Number \rightarrow sign \quad list$
  $sign \rightarrow +|-$
  $list \rightarrow list \quad bit|bit$
  $bit \rightarrow 0|1$

Example

- Consider a grammar for signed binary numbers
  $Number \rightarrow sign \quad list$
  $sign \rightarrow +|-$
  $list \rightarrow list \quad bit|bit$
  $bit \rightarrow 0|1$
- Build attribute grammar that annotates Number with the value it represents

Example

- Consider a grammar for signed binary numbers
  $Number \rightarrow sign \quad list$
  $sign \rightarrow + | -$
  $list \rightarrow list \quad bit | bit$
  $bit \rightarrow 0 | 1$
- Build attribute grammar that annotates Number with the value it represents
- Associate attributes with grammar symbols

Example

- Consider a grammar for signed binary numbers
  $Number \rightarrow sign \quad list$
  $sign \rightarrow +|-$
  $list \rightarrow list \quad bit|bit$
  $bit \rightarrow 0|1$
- Build attribute grammar that annotates Number with the value it represents
- Associate attributes with grammar symbols

Symbol	Attribute
number	value
sign	negative
list	position, value
bit	position, value

Attribute Rules

Production	Attribute Rule

Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	$list.position \leftarrow 0$
	`if sign.negative`
	`then number.value` $\leftarrow$ `- list.value`
	`else number.value` $\leftarrow$ `list.value`

Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	*list.position* $\leftarrow$ 0
	`if sign.negative`
	`then number.value` $\leftarrow$ `- list.value`
	`else number.value` $\leftarrow$ `list.value`

sign $\rightarrow +$	sign.negative $\leftarrow$ false

Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	$\mathit{list.position} \leftarrow 0$
	`if sign.negative`
	`then number.value` $\leftarrow$ `- list.value`
	`else number.value` $\leftarrow$ `list.value`

sign $\rightarrow +$	sign.negative $\leftarrow$ false
sign $\rightarrow$ -	sign.negative $\leftarrow$ true

Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	*list.position* $\leftarrow$ 0
	`if sign.negative`
	`then number.value` $\leftarrow$ `- list.value`
	`else number.value` $\leftarrow$ `list.value`

sign $\rightarrow$ +	sign.negative $\leftarrow$ false
sign $\rightarrow$ -	sign.negative $\leftarrow$ true
list $\rightarrow$ bit	bit.position $\leftarrow$ list.position list.value $\leftarrow$ bit.value

Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	$list.position \leftarrow 0$
	`if sign.negative`
	`then number.value ← - list.value`
	`else number.value ← list.value`

sign $\rightarrow$ +	sign.negative $\leftarrow$ false
sign $\rightarrow$ -	sign.negative $\leftarrow$ true
list $\rightarrow$ bit	bit.position $\leftarrow$ list.position list.value $\leftarrow$ bit.value
$list_0 \rightarrow list_1\, bit$	$list_1.position \leftarrow list_0.position + 1$ $bit.position \leftarrow list_0.position$ $list_0.value \leftarrow list_1.value + bit.value$

Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	$list.position \leftarrow 0$
	`if sign.negative`
	`then number.value ← - list.value`
	`else number.value ← list.value`

sign $\rightarrow +$	sign.negative $\leftarrow$ false
sign $\rightarrow$ -	sign.negative $\leftarrow$ true
list $\rightarrow$ bit	bit.position $\leftarrow$ list.position list.value $\leftarrow$ bit.value
$list_0 \rightarrow list_1\, bit$	$list_1.position \leftarrow list_0.position + 1$ $bit.position \leftarrow list_0.position$ $list_0.value \leftarrow list_1.value + bit.value$
bit $\rightarrow$ 0	bit.value $\leftarrow$ 0

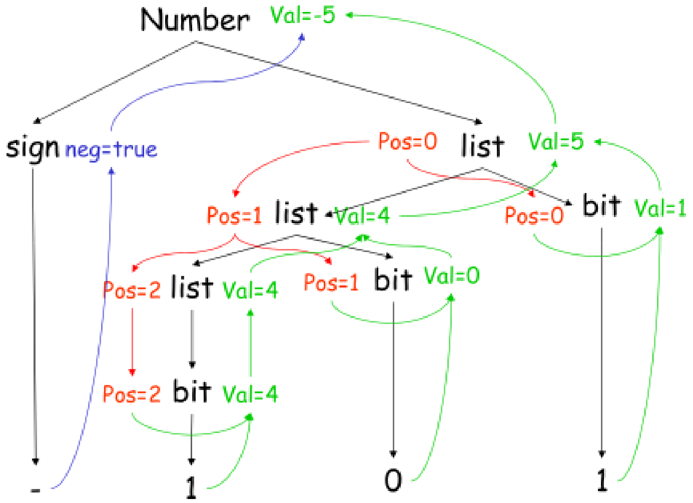
Attribute Rules

Production	Attribute Rule

number $\rightarrow$ sign list	$list.position \leftarrow 0$
	`if sign.negative`
	`then number.value ← - list.value`
	`else number.value ← list.value`

sign $\rightarrow +$	sign.negative $\leftarrow$ false
sign $\rightarrow$ -	sign.negative $\leftarrow$ true
list $\rightarrow$ bit	bit.position $\leftarrow$ list.position list.value $\leftarrow$ bit.value
$list_0 \rightarrow list_1\, bit$	$list_1.position \leftarrow list_0.position + 1$ $bit.position \leftarrow list_0.position$ $list_0.value \leftarrow list_1.value + bit.value$
bit $\rightarrow$ 0	bit.value $\leftarrow$ 0
bit $\rightarrow$ 1	bit.value $\leftarrow 2^{bit.position}$

Parse tree and the dependence graph


Number
Val=-5
sign
neg=true
Pos=0
list
Val=5
Pos=1
list
Val=4
Pos=0
bit
Val=1
Pos=2
list
Val=4
Pos=1
bit
Val=0
Pos=2
bit
Val=4
-
1
0
1