HW #1: Resolution Method and Parsing

Stepan Kuznetsov

Discrete Math Bridging Course, HSE University

Satisfiability

- We continue discussing satisfiability of Boolean formula.

Satisfiability

- We continue discussing **satisfiability** of Boolean formula.
- A satisfying assignment is an assignment of 0's and 1's to variables, which makes the formula true (value = 1).

Satisfiability

- We continue discussing **satisfiability** of Boolean formula.
- A satisfying assignment is an assignment of 0's and 1's to variables, which makes the formula true (value = 1).
- Satisfiability is a model example of a very general situation of **finding** (more precisely: checking for existence) an object with given properties.

Resolution Method

- Recall that **resolution method** is a method of determining whether a Boolean formula given in CNF is satisfiable.

Resolution Method

- Recall that **resolution method** is a method of determining whether a Boolean formula given in CNF is satisfiable.
- A **CNF** is a conjunction of clauses, where each clause is a disjunction of literals (e.g., $\overline{x} \vee y \vee \overline{z}$).

Resolution Method

- Recall that **resolution method** is a method of determining whether a Boolean formula given in CNF is satisfiable.
- A **CNF** is a conjunction of clauses, where each clause is a disjunction of literals (e.g., $\overline{x} \vee y \vee \overline{z}$).
- The algorithm **saturates** the CNF by adding all clauses which can be generated by the **resolution rule:**

$$\frac{A \vee p \quad B \vee \overline{p}}{A \vee B}$$

Resolution Method

- If the empty clause ($\perp$) got obtained, the CNF is not satisfiable (because the resolution rule keeps validity).

Resolution Method

- If the empty clause ($\perp$) got obtained, the CNF is not satisfiable (because the resolution rule keeps validity).
- Moreover, by **completeness theorem** this is a criterion: if the empty clause is not obtained, the CNF **is** satisfiable.

Resolution Method

- However, the non-derivability of the empty clause does not give us the satisfying assignment itself.

Resolution Method

- However, the non-derivability of the empty clause does not give us the satisfying assignment itself.
- In other words, the method solves the **decision problems** (“yes”/“no”), but not the **search problem.**

Resolution Method

- However, the non-derivability of the empty clause does not give us the satisfying assignment itself.
- In other words, the method solves the **decision problems** (“yes”/“no”), but not the **search problem.**
- If we are lucky enough, and the CNF has only one satisfying assignment, then after saturation we get **isolated** literals (like x or $\overline{y}$, for example), which dictate the desired satisfying assignment (e.g., $x = 1$ or $y = 0$).

Resolution Method

- In other cases, we can use the following consideration.

Resolution Method

- In other cases, we can use the following consideration.

Proposition

If a saturated CNF $\mathcal{S}$ includes neither $\perp$ nor $\overline{x}$ as an isolated literal, then $\mathcal{S} \wedge x$ is also satisfiable. Same for swapping x and $\overline{x}$.

Resolution Method

- In other cases, we can use the following consideration.

Proposition

If a saturated CNF $\mathcal{S}$ includes neither $\perp$ nor $\overline{x}$ as an isolated literal, then $\mathcal{S} \wedge x$ is also satisfiable. Same for swapping x and $\overline{x}$.

- In particular, if $\mathcal{S}$ is satisfiable and includes neither x nor $\overline{x}$, we can make an **arbitrary choice** for the value of x.

Resolution Method

- However, after making this arbitrary choice, we have to saturate $\mathcal{S} \wedge x$ (or $\mathcal{S} \wedge \overline{x}$) **again** before considering another variable.

Resolution Method

- However, after making this arbitrary choice, we have to saturate $\mathcal{S} \wedge x$ (or $\mathcal{S} \wedge \overline{x}$) **again** before considering another variable.
- For example, the CNF $(x \vee \overline{y}) \wedge (x \vee z)$ is saturated, but choosing $x = 0$ (adding $\overline{x}$) allows new resolutions giving $\overline{y}$ and z, and thus dictating values for all other variables.

Resolution Method

Proposition

If a saturated CNF $\mathcal{S}$ includes neither $\bot$ nor $\overline{x}$ as an isolated literal, then $\mathcal{S} \wedge x$ is also satisfiable. Same for swapping x and $\overline{x}$.

Resolution Method

Proposition

If a saturated CNF $\mathcal{S}$ includes neither $\perp$ nor $\overline{x}$ as an isolated literal, then $\mathcal{S} \wedge x$ is also satisfiable. Same for swapping x and $\overline{x}$.

- The proof of the proposition is easy.

Resolution Method

Proposition

If a saturated CNF $\mathcal{S}$ includes neither $\bot$ nor $\overline{x}$ as an isolated literal, then $\mathcal{S} \wedge x$ is also satisfiable. Same for swapping x and $\overline{x}$.

- The proof of the proposition is easy.
- Indeed, new resolutions applied when we saturate $\mathcal{S} \wedge x$, should involve x.

Resolution Method

Proposition

If a saturated CNF $\mathcal{S}$ includes neither $\perp$ nor $\overline{x}$ as an isolated literal, then $\mathcal{S} \wedge x$ is also satisfiable. Same for swapping x and $\overline{x}$.

- The proof of the proposition is easy.
- Indeed, new resolutions applied when we saturate $\mathcal{S} \wedge x$, should involve x.
- Therefore, if such a resolution generates $\perp$, there should have been $\overline{x}$ in the original $\mathcal{S}$.

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), p$$

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), p$$

$$(r \vee s), (\overline{s} \vee z), (\overline{p} \vee s \vee q), (\overline{p} \vee r \vee z), (\overline{s} \vee \overline{p} \vee t),$$

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), p$$

$$(r \vee s), (\overline{s} \vee z), (\overline{p} \vee s \vee q), (\overline{p} \vee r \vee z), (\overline{s} \vee \overline{p} \vee t),$$
$$(r \vee z), (\overline{p} \vee q \vee z), (s \vee q), (\overline{s} \vee t), (r \vee \overline{p} \vee t),$$

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), p$$

$$(r \vee s), (\overline{s} \vee z), (\overline{p} \vee s \vee q), (\overline{p} \vee r \vee z), (\overline{s} \vee \overline{p} \vee t),$$
$$(r \vee z), (\overline{p} \vee q \vee z), (s \vee q), (\overline{s} \vee t), (r \vee \overline{p} \vee t),$$
$$(q \vee t), (z \vee q), (r \vee t)$$

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), p$$

$$(r \vee s), (\overline{s} \vee z), (\overline{p} \vee s \vee q), (\overline{p} \vee r \vee z), (\overline{s} \vee \overline{p} \vee t),$$
$$(r \vee z), (\overline{p} \vee q \vee z), (s \vee q), (\overline{s} \vee t), (r \vee \overline{p} \vee t),$$
$$(q \vee t), (z \vee q), (r \vee t)$$

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), p$$

$$(r \vee s), (\overline{s} \vee z), (\overline{p} \vee s \vee q), (\overline{p} \vee r \vee z), (\overline{s} \vee \overline{p} \vee t),$$
$$(r \vee z), (\overline{p} \vee q \vee z), (s \vee q), (\overline{s} \vee t), (r \vee \overline{p} \vee t),$$
$$(q \vee t), (z \vee q), (r \vee t)$$

s

Example

$$(\overline{p} \vee r \vee s), (\overline{r} \vee q), (\overline{s} \vee \overline{p} \vee z), (\overline{z} \vee t), {\color{blue}p}$$

$$(r \vee s), (\overline{s} \vee z), (\overline{p} \vee s \vee q), (\overline{p} \vee r \vee z), (\overline{s} \vee \overline{p} \vee t),$$
$$(r \vee z), (\overline{p} \vee q \vee z), (s \vee q), (\overline{s} \vee t), (r \vee \overline{p} \vee t),$$
$$(q \vee t), (z \vee q), (r \vee t)$$

$${\color{red}s}, {\color{blue}z}, {\color{blue}t}, (\overline{p} \vee z), (\overline{p} \vee t)$$

$$\ldots$$

Resolution for 2-CNF

- If clauses include at least 3 literals, resolution can lead to growth:

$$\frac{x \vee \overline{y} \vee p \qquad z \vee w \vee \overline{p}}{x \vee \overline{y} \vee z \vee w}$$

Resolution for 2-CNF

- If clauses include at least 3 literals, resolution can lead to growth:

$$\frac{x \vee \overline{y} \vee p \quad z \vee w \vee \overline{p}}{x \vee \overline{y} \vee z \vee w}$$

- This makes saturation a potentially exponential procedure.

Resolution for 2-CNF

- If clauses include at least 3 literals, resolution can lead to growth:

$$\frac{x \vee \overline{y} \vee p \qquad z \vee w \vee \overline{p}}{x \vee \overline{y} \vee z \vee w}$$

- This makes saturation a potentially exponential procedure.
- However, for 2-CNF (each clause includes no more than 2 literals) the clauses do not grow:

$$\frac{x \vee p \qquad \overline{z} \vee \overline{p}}{x \vee \overline{z}}$$

Resolution for 2-CNF

- Thus, the total number of possible clauses does not exceed $4n^2 + 2n + 1$, where n is the number of variables.

Resolution for 2-CNF

- Thus, the total number of possible clauses does not exceed $4n^2 + 2n + 1$, where n is the number of variables.
- This makes the saturation process **polynomial.**

Resolution for 2-CNF

- Thus, the total number of possible clauses does not exceed $4n^2 + 2n + 1$, where n is the number of variables.
- This makes the saturation process **polynomial.**
- This can be organized as follows: take each clause from the list, starting from the second one, and try to resolve it against eariler ones. Does it give a new clause?

Resolution for 2-CNF

- Thus, the total number of possible clauses does not exceed $4n^2 + 2n + 1$, where n is the number of variables.
- This makes the saturation process **polynomial.**
- This can be organized as follows: take each clause from the list, starting from the second one, and try to resolve it against eariler ones. Does it give a new clause?
- New clauses are added to the bottom of the list.

Home Assignment # 1

- Satisfiability for 2-CNF will be your task for HW # 1.
- The **easy** version is to check satisfiability (using resolution method).
- The **full** task is to check satisfiability **and,** if the answer is "yes," to return one of the satisfying assignments.

Home Assignment # 1

- It is important to keep in mind that the input is given **in human-readable form,** as a string representing the formula.

Home Assignment # 1

- It is important to keep in mind that the input is given **in human-readable form,** as a string representing the formula.
- The program (in Python) should implement two functions:
  1. `is_satisfiable`, which takes a CNF and answers `True` or `False`, depending on whether it is satisfiable.
  2. `sat_assignment`, which takes a CNF and returns a satisfying assignment as an associative array:

     `{ 'x': True, 'y': False, 'z': True }`

Home Assignment # 1

- Conjunction, disjunction, negation, and implication, are, resp., `/\`, `\/`, `~`, `->`.

Home Assignment # 1

- Conjunction, disjunction, negation, and implication, are, resp., `/\`, `\/`, `~`, `->`.
- Literals: `x` or `~x`, where `x` is an arbitrary letter.

Home Assignment # 1

- Conjunction, disjunction, negation, and implication, are, resp., `/\`, `\/`, `~`, `->`.
- Literals: `x` or `~x`, where `x` is an arbitrary letter.
- Clauses: $(L_1$ `\/` $L_2)$ or $(L_1$ `->` $L_2)$, where L_1 and L_2 are literals.

Home Assignment # 1

- Conjunction, disjunction, negation, and implication, are, resp., `/\`, `\/`, `~`, `->`.
- Literals: `x` or `~x`, where `x` is an arbitrary letter.
- Clauses: $(L_1$ `\/` $L_2)$ or $(L_1$ `->` $L_2)$, where L_1 and L_2 are literals.
- The CNF is a conjunction (`/\`) of clauses.

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form (this is called **parsing** of the input).

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form (this is called **parsing** of the input).
- Grammar for CNFs:

```
CNF ::= Clause | CNF /\ Clause

Clause ::= (Lit \/ Lit) | (Lit -> Lit)

Lit ::= Var | ~Var
```

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form (this is called **parsing** of the input).
- Grammar for CNFs:

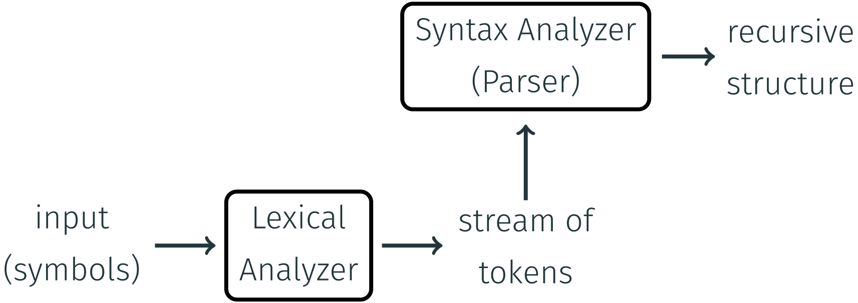
```
CNF ::= Clause | CNF /\ Clause

Clause ::= (Lit \/ Lit) | (Lit -> Lit)

Lit ::= Var | ~Var
```

- We shall use specialized software, PLY (Python Lex & Yacc), in order to automatize the parsing process.

The Parsing Workflow



Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

```
KW_INT
```

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

`KW_INT` `IDENT('main')`

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

`KW_INT`   `IDENT`(‘`main`’)   ‘`(`’

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

```
KW_INT    IDENT('main')    '('    KW_VOID
```

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

KW_INT   IDENT('main')   '('   KW_VOID   ...

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

`KW_INT` `IDENT`(‘`main`’) ‘(’ `KW_VOID` ...

- Tokens are much more convenient to work with (in the grammar).

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

$$(2x + 2)(3x^2 - 1) + 2x = 6x^3 + 6x - 2$$

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

$$(2x+2)(3x^2-1)+2x = 6x^3+6x-2$$

- Grammar:

```
Expr    ::= Tm | -Tm | Expr + Tm | Expr - Tm
Tm      ::= Mon | (Expr) | Tm (Expr)
Mon     ::= Int_opt 'x' Pow_opt | INT
Int_opt ::= INT | ε
Pow_opt ::= '^' INT | ε
```

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

$$(2x+2)(3x^2-1)+2x = 6x^3+6x-2$$

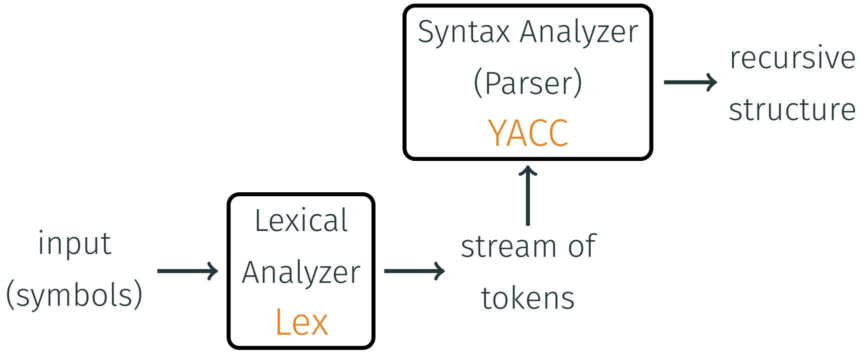
- Grammar:

```
Expr    ::= Tm | -Tm | Expr + Tm | Expr - Tm
Tm      ::= Mon | (Expr) | Tm (Expr)
Mon     ::= Int_opt 'x' Pow_opt | INT
Int_opt ::= INT | ε
Pow_opt ::= '^' INT | ε
```

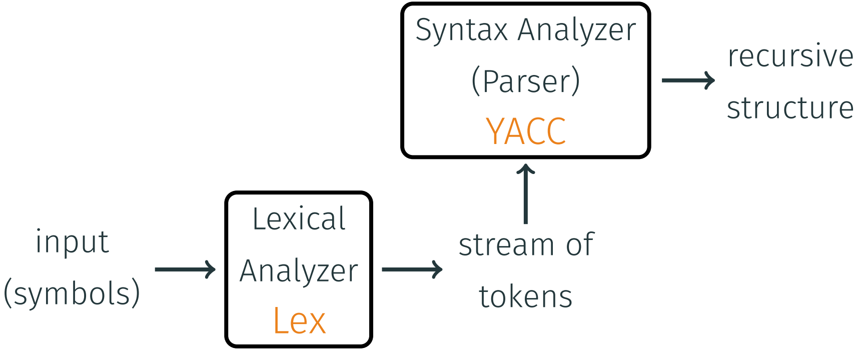
- Input example:

```
(2x+2)(3x^2-1)+2x
```

Implementation: Lex & Yacc

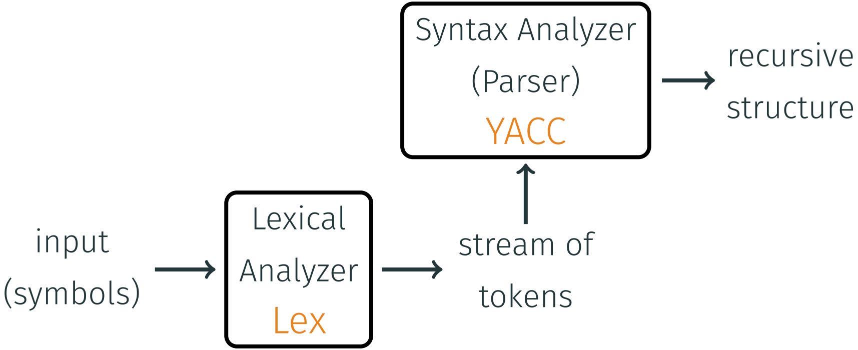


Implementation: Lex & Yacc



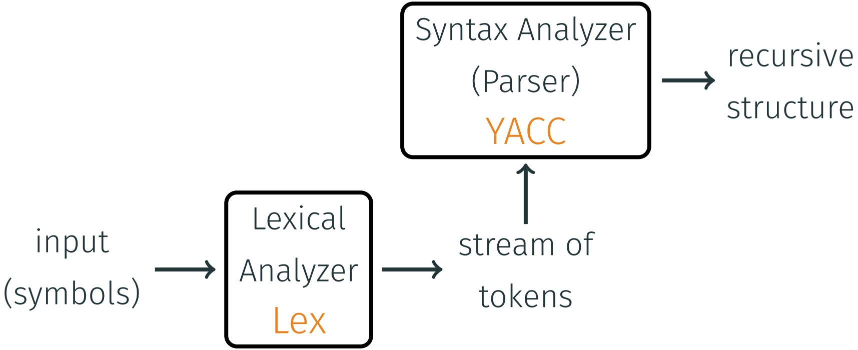
- YACC = Yet Another Compiler Compiler

Implementation: Lex & Yacc



- YACC = Yet Another Compiler Compiler

Implementation: Lex & Yacc



- YACC = Yet Another Compiler Compiler
- In Python, we use PLY (Python Lex & Yacc).

PLY Code for Lexical Analysis

- Declare tokens and literals (one-symbol tokens):

```
tokens = [ 'INT' ]
literals = ['+','-','(',')','^','x']
```

PLY Code for Lexical Analysis

- Declare tokens and literals (one-symbol tokens):

```
tokens = [ 'INT' ]
literals = ['+','-','(',')','^','x']
```

- For each token, declare a "`t_`"-function:

```
def t_INT(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print "Too large!", t.value
        t.value = 0
    return t
```

PLY Code for Lexical Analysis

- `r'\d+'` is a regular expression for sequences of decimal numbers.

PLY Code for Lexical Analysis

- `r'\d+'` is a **regular expression** for sequences of decimal numbers.
- Another example: regular expression for **names** (identifiers)

```
t_NAME     = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

PLY Code for Lexical Analysis

- `r'\d+'` is a **regular expression** for sequences of decimal numbers.
- Another example: regular expression for **names** (identifiers)

```
t_NAME    = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

- Finally, build the lexer:

```
import ply.lex as lex
lex.lex()
```

PLY Code for Parsing

- Each rule of the grammar is implemented as a "`p_`"-function:

```
def polymult(p,q) :
    r = []
    for i in xrange(len(p)) :
        for j in xrange(len(q)) :
            safeadd(r,i+j,p[i]*q[j])
    return r
```

...

```
def p_tm_mult(p):
    "tm : tm '(' expr ')'"
    p[0] = polymult(p[1],p[3])
```

PLY Code for Parsing

```
def p_tm_mult(p):
    "tm : tm '(' expr ')'"
    p[0] = polymult(p[1],p[3])
```

PLY Code for Parsing

```
def p_tm_mult(p):
    "tm : tm '(' expr ')'"
    p[0] = polymult(p[1],p[3])
```

- A “`p_`”-function generates an object `p[0]`, using `p[1]`, `p[2]`, ..., which are obtained from the lexer or recursively from parsing.

PLY Code for Parsing

- Finally, build the parser:

```
import ply.yacc as yacc
yacc.yacc()
```

PLY Code for Parsing

- Finally, build the parser:

```
import ply.yacc as yacc
yacc.yacc()
```

- The code of PLY examples is available on the course's webpage:

  https://homepage.mi-ras.ru/~sk/lehre/dm_hse/

PLY Code for Parsing

- Finally, build the parser:

```
import ply.yacc as yacc
yacc.yacc()
```

- The code of PLY examples is available on the course's webpage:

  `https://homepage.mi-ras.ru/~sk/lehre/dm_hse/`

- For priorities of operations, see another example available on the webpage: calculator.

Good luck!

Resolution: Completeness Proof

Theorem

If one cannot obtain the empty clause by applying resolutions, starting from the given CNF, then the CNF is satisfiable.

Resolution: Completeness Proof

Theorem

If one cannot obtain the empty clause by applying resolutions, starting from the given CNF, then the CNF is satisfiable.

- We prove this theorem using **induction** on the number of variables.

Resolution: Completeness Proof

Theorem

If one cannot obtain the empty clause by applying resolutions, starting from the given CNF, then the CNF is satisfiable.

- We prove this theorem using **induction** on the number of variables.
- That is, we establish it for zero variables (trivial) and then validate the **step** from n to $n+1$ variables.

Resolution: Completeness Proof

- Zero variables: the only possible clause is $\perp$, therefore, our CNF is empty.

Resolution: Completeness Proof

- Zero variables: the only possible clause is $\perp$, therefore, our CNF is empty.
- From n to $n+1$. Let the extra variable be $p_{n+1} = q$ and let $\mathcal{S}$ denote the saturation of our CNF.

Resolution: Completeness Proof

- Zero variables: the only possible clause is $\perp$, therefore, our CNF is empty.
- From n to $n+1$. Let the extra variable be $p_{n+1} = q$ and let $\mathcal{S}$ denote the saturation of our CNF.
- Take all clauses which do not include $\overline{q}$, and remove q out of them. This gives $\mathcal{S}^+$.

Resolution: Completeness Proof

- Zero variables: the only possible clause is $\perp$, therefore, our CNF is empty.
- From n to $n+1$. Let the extra variable be $p_{n+1} = q$ and let $\mathcal{S}$ denote the saturation of our CNF.
- Take all clauses which do not include $\overline{q}$, and remove q out of them. This gives $\mathcal{S}^+$.
- Dually, take clauses without q and remove $\overline{q}$. This gives $\mathcal{S}^-$.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.
  - Indeed, any new resolution in $\mathcal{S}^+$ or in $\mathcal{S}^-$ would induce a resolution in $\mathcal{S}$.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.
  - Indeed, any new resolution in $\mathcal{S}^+$ or in $\mathcal{S}^-$ would induce a resolution in $\mathcal{S}$.
- Let us show that **at least one** of $\mathcal{S}^+$ and $\mathcal{S}^-$ is satisfiable.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.
    - Indeed, any new resolution in $\mathcal{S}^+$ or in $\mathcal{S}^-$ would induce a resolution in $\mathcal{S}$.
- Let us show that **at least one** of $\mathcal{S}^+$ and $\mathcal{S}^-$ is satisfiable.
    - Suppose, both $\mathcal{S}^+$ and $\mathcal{S}^-$ include $\perp$.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.
  - Indeed, any new resolution in $\mathcal{S}^+$ or in $\mathcal{S}^-$ would induce a resolution in $\mathcal{S}$.
- Let us show that **at least one** of $\mathcal{S}^+$ and $\mathcal{S}^-$ is satisfiable.
  - Suppose, both $\mathcal{S}^+$ and $\mathcal{S}^-$ include $\perp$.
  - Then $\mathcal{S}$ includes both q and $\overline{q}$, and therefore $\perp$. Contradiction.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.
  - Indeed, any new resolution in $\mathcal{S}^+$ or in $\mathcal{S}^-$ would induce a resolution in $\mathcal{S}$.
- Let us show that **at least one** of $\mathcal{S}^+$ and $\mathcal{S}^-$ is satisfiable.
  - Suppose, both $\mathcal{S}^+$ and $\mathcal{S}^-$ include $\perp$.
  - Then $\mathcal{S}$ includes both q and $\overline{q}$, and therefore $\perp$. Contradiction.
  - Since $\mathcal{S}^+$ and $\mathcal{S}^-$ use only $p_1, \ldots, p_n$, we already know our theorem for them.

Resolution: Completeness Proof

- Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are saturated.
  - Indeed, any new resolution in $\mathcal{S}^+$ or in $\mathcal{S}^-$ would induce a resolution in $\mathcal{S}$.
- Let us show that **at least one** of $\mathcal{S}^+$ and $\mathcal{S}^-$ is satisfiable.
  - Suppose, both $\mathcal{S}^+$ and $\mathcal{S}^-$ include $\perp$.
  - Then $\mathcal{S}$ includes both q and $\overline{q}$, and therefore $\perp$. Contradiction.
  - Since $\mathcal{S}^+$ and $\mathcal{S}^-$ use only $p_1, \dots, p_n$, we already know our theorem for them.
  - The one which does not include $\perp$ is satisfiable.

Resolution: Completeness Proof

- If $\mathcal{S}^+$ is satisfiable, take the satisfying assignment and let $q = 0$.

Resolution: Completeness Proof

- If $\mathcal{S}^+$ is satisfiable, take the satisfying assignment and let $q = 0$.
- Clauses without $\overline{q}$ are already satisfied via $\mathcal{S}^+$.

Resolution: Completeness Proof

- If $\mathcal{S}^+$ is satisfiable, take the satisfying assignment and let $q = 0$.
- Clauses without $\overline{q}$ are already satisfied via $\mathcal{S}^+$.
- Clauses with $\overline{q}$ are satisfied by $\overline{q} = 1$.

Resolution: Completeness Proof

- If $\mathcal{S}^+$ is satisfiable, take the satisfying assignment and let $q = 0$.
- Clauses without $\overline{q}$ are already satisfied via $\mathcal{S}^+$.
- Clauses with $\overline{q}$ are satisfied by $\overline{q} = 1$.
- Dually, if $\mathcal{S}^-$ is satisfiable, take $q = 1$.

Beyond Propositional: Predicate Logic

- Of course, Boolean (*propositional*) logic is too weak for many situations.

Beyond Propositional: Predicate Logic

- Of course, Boolean (*propositional*) logic is too weak for many situations.
- In order to allow richer expressive capabilities, more powerful logical languages were introduced.

Beyond Propositional: Predicate Logic

- Of course, Boolean (*propositional*) logic is too weak for many situations.
- In order to allow richer expressive capabilities, more powerful logical languages were introduced.
- One of those is **first-order predicate logic,** which is usually used to formalize mathematics.

Predicate Logic

- In predicate logic, we have **individual variables** which range over a domain.

Predicate Logic

- In predicate logic, we have **individual variables** which range over a domain.
- **Atomic formulae** are of the form $P(x, y, z, ...)$, where P is a **predicate symbol**.

Predicate Logic

- In predicate logic, we have **individual variables** which range over a domain.
- **Atomic formulae** are of the form $P(x, y, z, ...)$, where P is a **predicate symbol**.
- E.g., a two-argument P denotes a **binary relation** (say, $x < y$, written as $< (x, y)$).‘

Predicate Logic

- In predicate logic, we have **individual variables** which range over a domain.
- **Atomic formulae** are of the form $P(x, y, z, ...)$, where P is a **predicate symbol**.
- E.g., a two-argument P denotes a **binary relation** (say, $x < y$, written as $< (x, y)$).‘
- Besides propositional operations ($\rightarrow$, $\vee$, $\wedge$, $\neg$), there are **quantifiers** $\forall$ (forall) and $\exists$ (exists).

Predicate Logic: Example

$$\forall x \forall y (R(x, y) \to \exists z (R(x, z) \land R(z, y)))$$

Predicate Logic: Example

$$\forall x \forall y (x < y \to \exists z (x < z \land z < y))$$

Predicate Logic: Example

$$\forall x \forall y (x < y \rightarrow \exists z (x < z \land z < y))$$

- This formula expresses the *density* of the order.

Predicate Logic: Example

$$\forall x \forall y (x < y \rightarrow \exists z (x < z \wedge z < y))$$

- This formula expresses the *density* of the order.
- Its truth depends on the interpretation: e.g., it is true on $\mathbb{Q}$ (rational numbers), but false on $\mathbb{Z}$ (integers).

Predicate Logic: Example

$$\forall x \forall y (x < y \rightarrow \exists z (x < z \wedge z < y))$$

- This formula expresses the *density* of the order.
- Its truth depends on the interpretation: e.g., it is true on $\mathbb{Q}$ (rational numbers), but false on $\mathbb{Z}$ (integers).
- So, it is **satisfiable,** but not **universally true.**

Predicate Logic: Example

$$\forall x \forall y (x < y \rightarrow \exists z (x < z \wedge z < y))$$

- This formula expresses the *density* of the order.
- Its truth depends on the interpretation: e.g., it is true on $\mathbb{Q}$ (rational numbers), but false on $\mathbb{Z}$ (integers).
- So, it is **satisfiable,** but not **universally true.**
    - Again, universal truth and satisfiability are dual.

Algorithmic Issues

- Unfortunately, satisfiability in predicate logic (unlike Boolean logic) is algorithmically undecidable.

Algorithmic Issues

- Unfortunately, satisfiability in predicate logic (unlike Boolean logic) is **algorithmically undecidable.**
  - This means that there is theoretically no algorithm for solving it, even without any time constraints.

Algorithmic Issues

- Unfortunately, satisfiability in predicate logic (unlike Boolean logic) is **algorithmically undecidable.**
  - This means that there is theoretically no algorithm for solving it, even without any time constraints.
- This motivates studying **decidable fragments** of predicate logic, where we restrict its expressivity in order to gain decidability.

Algorithmic Issues

- Unfortunately, satisfiability in predicate logic (unlike Boolean logic) is **algorithmically undecidable.**
  - This means that there is theoretically no algorithm for solving it, even without any time constraints.
- This motivates studying **decidable fragments** of predicate logic, where we restrict its expressivity in order to gain decidability.
  - Toy example: predicate logic with only unary predicates.

Decidable Fragments

- More interesting examples include description logics used in formal ontologies (used in OWL, SNOMED CT etc).

Decidable Fragments

- More interesting examples include **description logics** used in **formal ontologies** (used in OWL, SNOMED CT etc).
- These systems are between propositional and predicate logics and are used in **knowledge representation.**

Decidable Fragments

- More interesting examples include **description logics** used in **formal ontologies** (used in OWL, SNOMED CT etc).
- These systems are between propositional and predicate logics and are used in **knowledge representation.**
- Knowledge bases extend relational databases by a richer, logically enhanced language of queries. (This requires, obviously, fast algorithms.)