Overview & Motivation

A propositional logic formula is built from boolean variables, the ∧ , ∨ , ¬ operators, and parentheses.
A formula is satisfiable if there exists an assignment to the variables occurring in the formula that makes the formula true.

Cook-Levin Theorem: SAT is NP-complete
Any problem in the class NP can be reduced to SAT
An efficient algorithm for general SAT solving could solve many other problems efficiently
SAT can be solved in polynomial time if and only if P = NP

SAT solver: algorithm and tool that solves the SAT problem
Input: a SAT instance
Output: a satisfying assignment, or UNSAT if no such assignment exists
NP completeness => can't expect SAT solver to be efficient in general
But modern SAT solvers do work very well on large practical examples

General approach: create lots of variables in order to model something complex in Boolean algebra. Then encode constraints and feed to a solver like Z3.
To prove something is true, you can prove that the opposite is UNSAT.

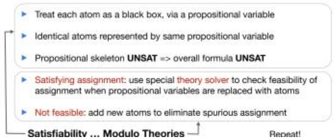
SMT-LIB 2 Declarations:

```
(declare-const x0 Bool)    (assert (= z0 (xor x0 y0)))
(declare-const x1 Bool)    (assert (= z1 (xor x1 y1)))
...
(declare-const y0 Bool)    (assert (= z7 (xor x7 y7)))
(declare-const y1 Bool)
...                        (assert (not
                             (or (and (not z0) w0)
                                 (and (= z0 w0)
                                   (or (and (not z1) w1)
                                     (and (= z1 w1)
                                       ...
```

N-queens example: encode each square as a variable (true is holds a queen). Then encode constraints as many xor. Also need to require queens: one on each row is a propositional variable

CMBC: tries to find a bug in C code via SAT solving (tries to find a case where some UB holds). Formula represents all possible executions. Used by AWS. Often needs to be given an unroll depth in order to produce a finite formula.

SAT: a formula is a boolean combination of propositional variables
SMT: a formula is a boolean combination of atoms



SMT Examples

```
(set-logic QF_NIA)

(declare-const circle Int)
(declare-const square Int)
(declare-const triangle Int)

(assert (= (+ circle circle) 10))
(assert (= (+ (* circle square) square) 12))
(assert (= (- (* circle square) (* triangle circle)) circle))

(check-sat)
(get-model)
```

Bit vectors:

```
(set-logic QF_BV)
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))

(assert (not (bvule (bvxor x y) (bvor x y))))

(check-sat)
```

SAT Solving

Naïve approach: truth tables. Try every possible assignment. However, this exponential complexity witch is not computable. Heuristics used to vastly improve performance in nearly all cases.

F ::= T | ⊥ | x | ¬F | F∧F | F∨F | F→F | F↔F

Literal: x and ¬x.

NNF (Negation Normal Form): No → or ↔ allowed. Negation can only be applied to variables. Double negation banned.
For example, $p \vee (\neg q \wedge (r \vee \neg s))$.
Conversion:

LE1: F1 → F2 ⇔ ¬F1 ∨ F2
LE2: F1 ↔ F2 ⇔ (¬F1 ∨ F2) ∧ (¬F2 ∨ F1)

LE3 (De Morgan's Law): ¬(F1 ∧ F2) ⇔ ¬F1 ∨ ¬F2
¬(F1 ∨ F2) ⇔ ¬F1 ∧ ¬F2

DNF (Disjunctive Normal Form): disjunction of conjuctions (∨ is outside of ∧). Each conjunction (a ∧ of many literals) is called a **clause**. Clauses cannot be negated.
Conversion:

First convert to **NNF**.
Distribute ∧ over ∨ via:
LE4: (F1 ∨ F2) ∧ F3 ⇔ (F1 ∧ F3) ∨ (F2 ∧ F3)
LE5: F1 ∧ (F2 ∨ F3) ⇔ (F1 ∧ F2) ∨ (F1 ∧ F3)

Satisfiability is trivial in DNF: any one clause can satisfy the formula. Therefore, you only need to find one clause that is satisfiable (i.e does not contain a literal and its negation). A satisfying assignment is just the literals is said formula, and the rest can be arbitrary.
However, distributing causes exponential increase in formula size: same issues as with naive approach.

CNF (Conjunctive Normal Form): conjunction of disjunction. Similar to **DNF**, but opposite. Each disjunction is now called a clause.
Conversion:
Same as above. NNF -> distribute. Lemmas are the same but with swapped symbols.

CNF also suffers from exponential blow-up of size, and satisfiability is not trivial. However, is the standard approach to SAT solving. This is due to algorithms to reduce growth to linear, and to make deductions about the formula that improve execution time.

Equisatisfiability: Two formulas are equisatisfiable if they have the same satisfiability. (both SAT or both UNSAT). Equivalent implies equisatisfiable.

Tseitin's transformation:
Will transform a formula into CNF with only linear blow-up in size. Preserves satisfiability, but not validity. The two formulas will be equisatisfiable.

Method:
Introduce a fresh variable for each sub-formula of F (including F itself) that isn't a variable (so also need new variables for a negated variable).

Example: if F is ((p ∨ q) ∧ r) → ¬s , we introduce 4 new variables:

p1 : representative of ¬s
p2 : representative of p ∨ q
p3 : representative of (p ∨ q) ∧ r
p4 : representative of ((p ∨ q) ∧ r) → ¬s

Each subformula variable (p) can be expressed as some operation on two others or variables (think: three-address code).
Generate these as formulae: i.e p2 ↔ p ∨ q
Then convert these to CNF using LE2.
Then take the conjuction of all of the above formulae (now in CNF), and also pF (where pF is the variable designated to the subformula of F that is just F).

The resulting F' is equisatisfiable to F, and is now is CNF with only linear growth in size.

Example Continued:

CNF(p1 ↔ ¬s) = (¬p1 ∨ ¬s) ∧ (s ∨ p1)
CNF(p2 ↔ p ∨ q) = (¬p2 ∨ p ∨ q) ∧ (¬p ∨ p2) ∧ (¬q ∨ p2)
CNF(p3 ↔ p2 ∧ r) = (¬p3 ∨ p2) ∧ (¬p3 ∨ r) ∧ (¬p2 ∨ ¬r ∨ p3)
CNF(p4 ↔ (p3 → p1)) = (¬p4 ∨ ¬p3 ∨ p1) ∧ (p3 ∨ p4) ∧ (¬p1 ∨ p4)

Example finalised:
F: ((p ∨ q) ∧ r) → ¬s is equisatisfiable to:
p4 ∧
(¬p1 ∨ ¬s) ∧ (s ∨ p1) ∧
(¬p2 ∨ p ∨ q) ∧ (¬p ∨ p2) ∧ (¬q ∨ p2) ∧
(¬p3 ∨ p2) ∧ (¬p3 ∨ r) ∧ (¬p2 ∨ ¬r ∨ p3) ∧
(¬p4 ∨ ¬p3 ∨ p1) ∧ (p3 ∨ p4) ∧ (¬p1 ∨ p4)

The number of subformulae is bound by the number of variables in F (in the worst case, each new variable introduces one new subformula). Each subformula can only have 3 variables, and therefore the CNF has a constant maximum size. Therefore, the conversion is O(n).

Once in CNF, a SAT solving algorithm can be used such as **DPLL**. Then, the **model** (satisfying assignment) can be mapped to F by dropping the assignments to the extra variables introduced by Tseitin.

NP-Completeness: SAT solving is NP-complete. 3-SAT is a similar problem where all clauses can only have exactly 3 literals. Can map from SAT input to 3-SAT input, so SAT reduces to 3-SAT. Therefore, 3-SAT is NP-complete. 3-SAT is sometimes more useful for proving other NP-complete problems.
Proving NP-completeness: show that some known NP-complete problem reduces to the problem you are trying to prove (to reduce to P, you need to be able to map your input in P-time such that your problem can be solved by P). Essentially, you need to be able to solve a NP-complete problem with only a polynomial time modification of its input.

DPLL

Due to Davis, Putnam, Loveland, Logemann, 1962. Combines search and deduction (logical inference) to solve efficiently. Only accepts formula in **CNF**.

A **unit clause** is a clause containing only one literal. An **empty clause** contains no literals, and is equivalent to ⊥.

Rule 1: If clause C1 contains p, and C2 contains ¬p, then they can be merged into C3 which is just the disjunction of all the other literals of C1 and C2. This is because if C3 is satisfied, then one of C1 and C2 must be as well, and we can always assign p to satisfy the other.

Unit resolution: If a unit clause exists, it can always be satisfied. Therefore, it can be removed from the formula. Any other clause that contains that literal can also be removed as it is now satisfied. That literal's negation can be removed from any clauses because we know that literal is now assigned as ⊥. This may lead to new unit clauses, which could then be resolved. This process is formalized as **Boolean Constraint Propagation (BCP)**. This can also lead to empty clauses. In this case we return UNSAT and terminate. This is because we have reached a contradiction, where we have assigned every literal in a clause to ⊥ due to unit resolution. Therefore, that clause is unsatisfiable, and so must be the formula.

Pure literal propagation (PLP): If a variable only appears positively (or negatively) in a formula, then we can assign that variable to true (or false) and immediately remove all clauses in which it appears.

DPLL Algorithm: (top level call: DPLL(F, []))

```
// Input: CNF formula F, variable assignment A
// Return: satisfying assignment of F if SAT, otherwise UNSAT
def DPLL(F,A):

  (A', F') = PLP(BCP(F))
  A = A ∪ A'   // Extend A with new assignments from BCP + PLP
  if F' == T:
    return A
  if F == ⊥:
    return UNSAT

  p = choose_var(F') // Select an unassigned variable

  maybeA = DPLL(F'[p -> T], A[p -> true]) // Recurse
  if maybeA != UNSAT:
    return maybeA

  return DPLL(F'[p -> ⊥], A[p -> false]) // Recurse
```

DPLL Example

Worked example

$F: (\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$

- No unit clauses: BCP not possible
- No pure literals: PLP not possible

Choose a variable to branch on: q

$F[q \mapsto true] : (\neg p \vee \top \vee r) \wedge (\bot \vee r) \wedge (\bot \vee \neg r) \wedge (p \vee \bot \vee \neg r)$

Simplifies to:

$F[q \mapsto true] : (r) \wedge (\neg r) \wedge (p \vee \neg r)$

Worked example

$F[q \mapsto true] : (r) \wedge (\neg r) \wedge (p \vee \neg r)$

BCP: unit resolution for (r) and $(\neg r)$ deduces $\bot \Rightarrow$ **backtrack**

$F: (\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$

Now try $q = false$

$F[q \mapsto false] : (\neg p \vee \bot \vee r) \wedge (\top \vee r) \wedge (\top \vee \neg r) \wedge (p \vee \top \vee \neg r)$

Simplifies to:

$F[q \mapsto false] : (\neg p \vee r)$

Worked example

$F[q \mapsto false] : (\neg p \vee r)$

Apply **PLP**: both $\neg p$ and r are now pure literals

- Set p to $false$ and r to $true$

$F[q \mapsto false, p \mapsto false, r \mapsto true] : (\neg \bot \vee \top)$, simplifies to $\top$

Thus F is satisfiable, and:

$[q \mapsto false, p \mapsto false, r \mapsto true]$

is a satisfying assignment, a.k.a. a model, for F

CDCL – Conflict Driven Clause Learning

Upgraded DPLL. Differs in 3 main ways:
Non-chronological backtracking - can backtrack aggressively to avoid 'doomed' exploration
Learning: conflict clauses can be added to the formula to prevent similar dead ends in other parts of the resolution 'tree' from being explored.
Heuristics: Attempts to make good decisions when choosing variables to resolve from.

Partial assignments (p -> true, q -> false) can be thought of as a conjunction of unit clauses (p ∧ ¬q) in order to exploit BCP.

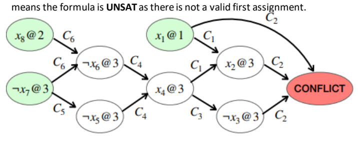
Changes to BCP: BCP now also takes a partial assignment which it extends with assigned that can be inferred via **unit resolution.** Does not return an edited formula; just CONFLICT or OK.

CDCL Algorithm:

```
// Input CNF formula F
// Returns: assignment A if F is satisfiable
            false otherwise
def CDCL(F) :
  A = {}
  if (BCP(F,A) == conflict) return false
  level = 0
  while (hasUnassignedVars(F,A)) :
    level = level + 1
    DECIDE(F,A) // Choose an unassigned variable
    while (BCP(F,A) == conflict) :
      (level, K) = ANALYSE_CONFLICT(F,A)
      F = F ∧ K
      if (level < 0) return false
      BACKTRACK(A, level)

  return A
```

An **Implication graph** can be built which describes which assignments lead to others, so that if a conflict is reached, we can learn something about the formula.
Decision levels are used to describe how deep into the resolution "tree" we are, so that the correct assignments can be removed when backtracking. The first decision is made at level 1. If the decision level falls below 0, that means the formula is **UNSAT** as there is not a valid first assignment.




```
while (BCP(F, A) == conflict) {
  (level, K) = ANALYSE_CONFLICT(F, A);
  F = F ∧ K;
  if (level < 0) return false;
  BACKTRACK(A, level);
}
return A;
```

$(1, \neg x_1 \vee \neg x_4)$

Backtrack to decision level 1 | Add this conflict clause to formula

Here, the implication graph tells us that because we assigned x1 and x4 to ⊤, a conflict occurred. Therefore, we know that either ¬x1 or ¬x4 needs to be ⊤ (¬x1 ∨ ¬x4 needs to be SAT). We then add this clause to our formula to help later BCP make more correct assignments. We backtrack to the second highest decision level (more discussion later) of any decision from our conflict clause, so BCP can run again with the new clause and without some of the assignments that led to the conflict. In some cases, we will need to backtrack further as that assignment may lead to a conflict by itself.
Learned **conflict clauses** are implied by the formula (by construction – the opposite caused a conflict so the clause must be implied). Smaller conflict clauses are preferred because they would block BCP progress earlier, increasing efficiency.

Pure literal propagation is generally not used in **CDCL**, as it is not efficient enough to spend computation time on.

To build a **conflict clause**, we need to **cut** the implication graph so that all paths from decisions to the conflict note are cut. Therefore, all decisions nodes are on one side of the cut (the **reason side**), and the conflict node on the other (the **conflict side**). Formally, the conflict clause is the disjunction of the negation of all the nodes adjacent to the cut on the **reason side**.

A (non-conflict) node is a **unique implication point (UIP)** if it is on every path from the latest decision literal (the one made at the highest decision level) to the conflict node. The **First UIP** is the UIP closest to the conflict node.

A good strategy for finding the best **separating cut** is to cut immediately after the **First UIP**, so that any nodes reachable by the first UIP are on the **conflict side**. As the first UIP is the closest to the conflict node, this is likely to cut less edges and therefore lead to a smaller **conflict clause**.

Intelligent **backtracking** is not required for termination; we could backtrack to level 0 every time, and due to conflict clauses, progress is still made.
Chronological backtracking (like **DPLL**), where we always just reduce the decision level by 1, would often lead to the same conflict. Therefore, as described earlier, we backtrack to the **second highest** decision level that led to the conflict (i.e we remove the assignment at the highest relevant decision level). This means that the conflict clause will be unit (as we are still assigning the opposite of all but one of the literals), and BCP can hopefully make further progress. If only one decision led to the conflict, we backtrack to decision level 0: again, we now have another unit clause for BCP from the conflict. We have determined that that single assignment would lead to a conflict, and thus its negation must be ⊤. Importantly, that assignment would be an **implied literal**, not a **decision literal**, so we can't remove it from backtracking: it is enforced by the formula.

It is possible, via conflict clauses, to reach a conflict at decision level 0. This is very similar to finding an empty clause in **DPLL**; the formula includes a contradiction. As all conflict clauses are implied by the original formula, that must have a contradiction also, so we should return UNSAT. The ANALYSE_CONFLICT function would return decision level -1 in order to signal UNSAT.

Decision Heuristics

When BCP stops making process, we pick a variable to assign and continue resolving. This decision is an issue of efficiency, not correctness; we want to explore the most useful parts of the resolution tree.

DLIS (Dynamic Largest Individual Sum): Choose the assignment that satisfies the most unsatisfied clauses (i.e that appears in the most clauses). As assignments and backtracking can change the remaining unsatisfied clauses, this information needs to be recalculated at every decision point. We can reduce the work slightly by keeping the counts for every decision level, and when editing those with the information from the conflict clause. This is still expensive but not nearly as bad. Only running the algorithm once at the start of execution would not be as effective, as adding conflict clauses can change the counts.

VSIDS (variable, state independent, decaying sum): at the start of execution, count how many clauses each literal occurs in. When a conflict clause is learned, increment the count of each literal in the clause. Scores are decayed (usually halved) periodically. Take the unassigned literal with the highest score. The decay means that literals appearing in recent conflict clauses are prioritised, as they are more likely to be relevant and lead to conflicts faster. VSIDS is efficient as the ongoing cost is minimal; you only need to scan over the clauses once.

Number of conflict clauses: in the worst case, the number of conflict clauses is exponential. This can lead to massive efficiency issues, especially for BCP which works by scanning over the clauses. We can remove **subsumed clauses** as they don't give us anything. For example, if we have (p ∨ q) then the clause (p ∨ q ∨ r) is **subsumed** by it. As soon as we satisfy the first clause, we satisfy the second, so the second clause is only wasting space and can be removed. Formally, we can remove any clause which is strictly a superset of another clause I the formula. This happens often when adding conflict clauses; if this conflict is at a lower decision level than previous ones, then it is likely that this clause will subsume some of those. Subsumption checking is expensive, however, and it is not always enough to deal with the size blow-up due to conflict clauses.
Many modern solvers place a limit on the number of conflict clauses stored; ideally, clauses relevant to the current search space are kept, so common algorithms are based on **VSIDS** score or age. However, there are obvious efficiency downsides and termination concerns.
Improving BCP: Many BCP implementations store a map from variables to the clauses they appear in; this allows the algorithm to easily find which clauses included the negation of the assigned variable and therefore may now be unit. However, this is not a silver bullet to the problem of formulae being very large, and BCP is still very expensive.
Watch literals: for each clause, we can assign two literals to be **watch literals**; unless those variables are assigned, we know this clause can never become unit. For each literal, a list of clauses it is watching is stored. When a watch literally is assigned we can inspect the clause' either it is now unit, or we can assign a new watch literal. This means that BCP does not need to scan the entire set of clauses at every assignment, because literals that aren't **watch literals** aren't currently relevant. This means that a new watch literal might already be assigned, but it is simple to continue iterating until an unassigned literal is found or the clause is unit.
Restarts: due to the "tree" structure of CDCL, the algorithm fully explores sub regions before backtracking far enough to reach new space. This can be inefficient so we implement occasional **restarts** to level 0, where the current assigned are destroyed. **VSIDS** (or similar heuristic) scores and conflict clauses are retained, as their information is globally relevant. The scores remain meaning the variables assigned at a high decision level previously are likely to be assigned early (at a lower level), meaning it is likely we are now exploring a different part of the tree. We are now likely to build smaller conflict clauses, because we are at a low level, which provide more information.
Watch literals making BCP efficient mean that restarts aren't that expensive' modern solvers restart as frequently as every 20 backtracks.

SMT Solving

Used to find satisfiability of first order logic problems, but with some meaning assigned to the atoms. We are interested in SAT solutions that also fit the constraints from the SMT theory.

SMT formula: built from atoms, using boolean connectives and quantifiers
Term: a variable, constant, or function applied to a sequence of terms
Atom: a predicate of the theory, applied to a sequence of terms
Literal: an atom or its negation
Theory determines the available constants, functions and predicates
Some SMT theories are decidable, some are only if they are used quantifier-free, and some are not.

Bit-vectors: An SMT theory. Variables are fixed-width bit vectors, with expected functions and predicates. Fully decidable. **Floating points** are similar, and include a **rounding mode** argument to every function. **Non-linear integer arithmetic** is always undecidable. Decidable if remove multiplication (linear integer arithmetic) but super-exponential with quantifiers. NP-complete without.

Solvers:
Z3: supports wide range of theories, including quantifiers
CVC (currently CVC5): supports wide range of theories, including quantifiers
Bitwuzla: specialised towards bit-vectors, floating-point, arrays and uninterpreted functions
Yices: supports real and integer arithmetic (linear and non-linear) and bitvectors

SMT general idea:
Use SAT solver to decide whether SMT formula might have a model, based on its structure F
Abstract to propositional logic formula encoding boolean structure of
Apply SAT solver to abstract form - if UNSAT, is UNSAT
Otherwise: use theory solver to check whether assignment produced by SAT solver is feasible
If feasible, is SAT
Otherwise: refine the abstract formula, and apply SAT solver again

Abstraction: just naively replace every atom with a boolean variable. This process is referred to as **B**.
Theory: linear integer arithmetic

Formula F: $\neg(x \geq 3) \wedge (x \geq 3 \vee x \geq 5)$

Boolean abstraction $B(F)$: $\neg p_1 \wedge (p_1 \vee p_2)$

Another formula ϕ in boolean abstraction: $\neg p_1 \wedge (p_1 \vee p_2) \wedge (p_1 \vee \neg p_2)$

Inverse boolean abstraction $B^{-1}(\phi)$:

$\neg(x \geq 3) \wedge (x \geq 3 \vee x \geq 5) \wedge (x \geq 3 \vee \neg(x \geq 5))$

Basic SMT algorithm

```
// Input: SMT formula F over theory T
// Assumption: theory solver for T is available
// Returns: either UNSAT, or a model for F
SolveSMT(F) {
  ϕ = B(F);
  while(true) {
    A = CDCL(ϕ); // Returns UNSAT or assignment
    if(A == UNSAT) return UNSAT;
    M = TheorySolve(B⁻¹(A)); // Returns UNSAT or model
    if(M != UNSAT) return M;
    ϕ = ϕ ∧ ¬A;
  }
}
```

Formula F: $\neg(x \geq 3) \wedge (x \geq 3 \vee x \geq 5)$

Boolean abstraction $\phi = B(F)$: $\neg p_1 \wedge (p_1 \vee p_2)$

CDCL(ϕ) yields satisfying assignment $\neg p_1 \wedge p_2$

Apply inverse boolean abstraction:

- $B^{-1}(\neg p_1 \wedge p_2) = \neg(x \geq 3) \wedge x \geq 5$

Ask theory solver "is this solution allowed by the theory?"

- Apply theory solver to conjunction of theory literals: $\neg(x \geq 3) \wedge x \geq 5$

Running example

Theory solver reports $\neg(x \geq 3) \wedge x \geq 5$ is **UNSAT**

- i.e. $x \geq 3 \vee \neg(x \geq 5)$ is valid

Add boolean abstraction of this theory lemma to abstract formula ϕ

- Called a theory conflict clause

Boolean abstraction ϕ becomes:

- $\neg p_1 \wedge (p_1 \vee p_2) \wedge \boxed{(p_1 \vee \neg p_2)}$

This is the theory conflict clause. Note that it is a propositional logic clause - it's called "theory" because we used the theory solver to find it

CDCL(ϕ) now returns **UNSAT** => original formula F is **UNSAT**

Assuming the **theory solver** is a decision procedure (i.e given well-formed formula in its theory, will yield a result), then this SMT algorithm always terminated (similarly to CDCL, we always make progress via theory clauses). However, extremely expensive because theory clauses from the theory solver rule out the exact assignment given by the SAT solver.
Solution: find the **minimal unsatisfiable core** of the assignment, and add that as a conflict clause instead. This can be done by removing assignments from the model and repeatedly calling the theory solver, until the have an UNSAT set of assignments that would lead to SAT if we remove any of them. This gives the most information to the SAT solver and greatly increases performance. Finding the minimal unsatisfiable core can be done in linear time.

This approach, where the SAT solver and theory solver are considered as black boxes, is called **offline** SMT solving. It is good for separation of concern and correctness, but not performance. **Online** SMT solving integrates the theory solver into CDCL, confusingly referred to as **DPLL(T)**. Can call the theory solver after every decision (and consequent BCP) using the partial assignments, so invalid assignments due to the theory can be reported to the SAT solver as soon as possible. A theory clause can then be learnt, and it will also be smaller than if we waited to find a satisfying assignment! Minimal satisfiable core approach is still used at this point. Then, BCP is ran again, which has to lead to a conflict, and we continue with a backtrack as normal. Importantly, theory clauses are implied by the theory, not the Boolean formula.

Using SMT

A simple C-like programming language

Data type: `int`

Operations:

- Usual integer operators (+, -, *, etc)
- Boolean operators to allow conditions: ==, <=, &&, ternary (?:), etc.

Statements:

- Assignment: $v = e$;
- Assertion: `assert(e);`
- Conditional: `if(e) [Stmt] else [Stmt]`
- Sequence of statements

These may be sequences of statements, containing nested conditionals

Also, variables are always assumed to exist and can only be 32 bit integers. A program is just a sequence of statements and side effects are banned.
Execution **terminates normally** if every assertion succeeds; **terminates with an error** otherwise. If all initial states lead to normal termination, then the program is **correct**.

```
y = x + 1;
assert(y > x);
```

Terminates abnormally when x is INT_MAX

```
y = x + 1;
if(x < 1000) {
  assert(y > x);
} else {
  assert(y != x);
}
```

Terminates normally for all inputs

```
y = x + 1;
if(y < 1000) {
  assert(y > x);
} else {
  assert(y != x);
}
```

Again, terminates abnormally when x is INT_MAX

We will use **bit-vectors** to transform our language into an SMT formula F, such that the program is correct if F is UNSAT. This means that there was no assignment where a bug occurs in the program. SSA is also necessary, as SMT requires variables to have a global value. Other theories can be used, but bit-vecs make sense in this integer world. SSA assignments can be translated into assert statements, describing the relationship between the variables. All assertions in the original program are negated, and then disjuncted so that only one of these negated asserted need to be true in order for the formula to be SAT, meaning the program is incorrect.

The above works if we don't have loops and function calls; loop need to be abstracted using invariants, and functions can either be inlined or verified modularly using pre-conditions and post-conditions.

Checking correctness of an SSA program

```
x = y + 1;         x1 = y0 + 1;
x = x + 1;         x2 = x1 + 1;
y = y + 1;         y1 = y0 + 1;
assert(x == y + 1);   assert(x2 == y1 + 1);
assert(x > y);        assert(x2 > y1);
```

Constraints satisfiable iff there exist values for x_1, x_2, y_0, y_1 that:

- Satisfy relationships between variables enforced by assignments
- Cause at least one assertion to fail

```
(assert (= x1 (+ y0 1)))
(assert (= x2 (+ x1 1)))
(assert (= y1 (+ y0 1)))
(assert (or (not (= x2 (+ y1 1)))
            (not (> x2 y1))))
```

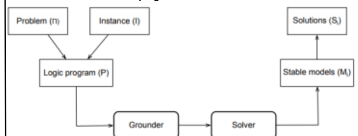
P correct iff constraints are UNSAT

Answer Set Programming

	Prolog	ASP
basic data structure	(nested) terms	(flat) terms
variables	unification	instantiation
computation	top-down query evaluation	bottom-up model finding
solution	yes/no, substitutions	answer sets

	SAT	ASP
language	propositional logic	logic programs + extensions
assumption	open world	closed and open world
reasoning	satisfiability	satisfiability, enumeration/projection, intersection/union, optimization
system	solving	modelling & solving
complexity	NP	NP and NP^NP for disjunctive extensions

The order of rules in the program does not matter in ASP



Relevant ground instantiation: for a rule Φ in the program, find the ground instantiations (cs). Then for each c, add if every atom that appears positively in the body also appears in the head of another relevant ground instantiation. Then iterate. Intuition: The ground instantiations that are actually reachable by the program. Necessary because the ground instantiation is infinite.
Example:
even(0).
odd(s(X)) <- even(X)

relevant ground instantiation = {even(0), odd(s(0)) <- even(0)}.

Safe rule: a rule is safe if all of it's variables appear in at least one a positive body literal.
ASP programs can also contain integrity constraints, choice rules, cardinality constraints, conditional literals, aggregate expressions, and optimisation statements.
Integrity constraints: eliminate unwanted stable models. Of the form <- a,b, …, not m, not n, … (for arbitrary body atoms)
In Clingo, written as :- …
Sugar for p <- a, b, …, not p (clauses a, b, … lead to a contradiction)
Constraint is **satisfied by M** any of the body literals are false in M.
Choice rule: {a,b,…} <- c,d,…
Head like this is called an **aggregate**
Choice rule is satisfied if **any subset** (even empty) of the clauses of the head are in M, and all of the body is too.