Programmierung 2 (SS 2020)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Projekt 2

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Tina Jung, B. Sc.
Maximilian Fickert, M. Sc.

SatSolver (20 Punkte)

In diesem Projekt ist Ihre Aufgabe, mit einem `C` Programm logische Formeln in konjunktiver Normalform einzulesen und auf ihre Erfüllbarkeit zu prüfen. Ein Hauptlernziel dieses Projektes ist, ein Programm von Grund auf selbst zu programmieren. Daher wird in diesem Projekt kein Code vorgegeben und Ihnen ist frei überlassen wie Sie ihr Programm strukturieren, also welche Datenstrukturen Sie nutzen, wie Sie Ihre Methoden nennen und welche `.c`- und `.h`-Dateien Sie erstellen. Damit Sie sich aber nicht in ihrer Freiheit verlieren, wird der zu implementierende Algorithmus genau spezifiziert. Weil wir keine Vorgaben für die interne Projektstruktur machen, wird das Projekt ausschließlich mit Integration Tests evaluiert. Achten Sie daher besonders darauf sich genau an Eingabe- und Ausgabeformate zu halten.
Im Folgenden werden zunächst das Erfüllbarkeitsproblem und der Algorithmus zur Lösung dieses Problems erläutert. Dann wird das zu schreibende Programm spezifiziert, dabei werden Eingabe- und Ausgabeformate definiert und es wird Ihre konkrete Aufgabenstellung angegeben. Im letzten Abschnitt finden Sie noch einige Hinweise zu Implementierungsdetails.

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT[1])

Gegeben einer aussagelogischen Formel wie zum Beispiel

$$(A \vee B \vee C) \wedge (\neg A \vee B) \wedge (\neg C), \tag{1}$$

wobei A, B und C logische Variablen sind, ist das Erfüllbarkeitsproblem die Frage, ob es eine Variablenbelegung gibt, sodass die Formel als *wahr* ausgewertet wird. Für dieses Projekt interessieren uns nur Formeln in *konjunktiver Normalform* (*KNF*, englisch: *conjunctive normal form*, *CNF*).

Konjunktive Normalform

Ein *Literal* ist eine nichtnegierte oder eine negierte Variable. Sei X eine Variable. Dann bezeichnen wir $\neg X$ als negatives Literal oder auch Literal mit negativer Polarität und X als positives Literal oder Literal mit positiver Polarität. Eine *Klausel* ist eine Disjunktion („Oder-Verknüpfung“) von endlich vielen Literalen. Eine aussagenlogische Formel ist in konjunktiver Normalform wenn sie eine Konjunktion („Und-Verknüpfung“) von endlich vielen Klauseln ist.
In dieser Darstellung gibt es also keine Operatoren für logische Implikation und Äquivalenz. Außerdem darf der Negationsoperator nur auf Variablen angewandt werden und nicht auf beliebige Formeln.
Die Formel in 1 ist in konjunktiver Normalform. In dem Beispiel sind $(A \vee B \vee C)$, $(\neg A \vee B)$ und $(\neg C)$ die Klauseln. Die positiv auftretenden Literale sind A, B und C, und die Literale negativer Polarität sind $\neg A$ und $\neg C$.

exact-3-SAT

Für $k \in \mathbb{N}$, ist eine Formel ϕ in *exact-k-CNF*, wenn ϕ in konjunktiver Normalform ist und jede Klausel in ϕ aus genau k Literalen besteht.
Beispiel 1 ist also nicht in exact-3-CNF, die folgende (äquivalente) Formel aber schon:

$$(A \vee B \vee C) \wedge (\neg A \vee B \vee B) \wedge (\neg C \vee \neg C \vee \neg C). \tag{2}$$

exact-3-SAT ist der Spezialfall des Erfüllbarkeitsproblems für Formeln in exact-3-CNF[2]. Ihre Aufgabe wird sein, ein Programm zu schreiben, welches das exact-3-SAT-Problem löst.

[1] „SAT“ kommt vom englischen Wort „satisfiability“, welches „Erfüllbarkeit“ bedeutet.
[2] Jede klassisch aussagelogische Formel lässt sich in eine äquivalente Formel in exact-3-CNF umwandeln.

Davis-Putnam-Logemann-Loveland-Algorithmus (DPLL)

Der DPLL-Algorithmus prüft für eine aussagenlogische Formel in konjunktiver Normalform, ob diese erfüllbar ist. Er sucht dazu systematisch nach einer Belegung der Variablen, die die Formel erfüllt. Die Grundversion des Algorithmus arbeitet wie folgt: Initial ist der Wahrheitswert jeder Variable *undefiniert*. Eine Variable wird ausgewählt und ihr wird der Wert *wahr* zugewiesen. Dann wird in einer Schleife geprüft, ob die verbleibenden Variablen noch so belegt werden können, dass die Formel erfüllt ist. Wenn ja, so ist die Ausgangsformel erfüllbar. Wenn nein, wird die ausgewählte Variable auf den Wert *falsch* gesetzt und es wird erneut nach einer gültigen Belegung der restlichen Variablen gesucht. Findet sich auch für den neuen Wert keine gültige Belegung, so ist die Ausgangsformel unerfüllbar. Zudem gibt es zwei Optimierungen (*Unit-Propagation* und *Pure-Literal-Elimination*), die in bestimmten Fällen erlauben, sofort einen Wert für eine Variable zu schließen. In jeder Iteration werden die folgenden Schritte ausgeführt:

Abbruch Wenn die Formel zu *wahr* auswertet, wird der Algorithmus abgebrochen. Die Formel ist erfüllbar. Wenn die Formel zu *falsch* auswertet und allen bisher gesetzten Variablen kein anderer Wahrheitswert zugeordnet werden kann, wird der Algorithmus abgebrochen. Die Formel ist unerfüllbar. Wenn der Wahrheitswert der Formel *undefiniert* ist, wird nicht zurückgesetzt und die Iteration wird fortgesetzt.

Rücksetzung Wenn die Formel zu *falsch* auswertet und es eine Variable gibt, für die ein alternativer Wert ausprobiert werden kann, werden die letzten Iterationen des Algorithmus rückgängig gemacht, bis die Variable wieder unbelegt ist. Dann wird die Variable mit dem alternativen Wert belegt und die nächste Iteration des Algorithmus wird begonnen.

Pure-Literal-Elimination Wenn es eine nicht belegte Variable gibt, welche in allen noch nicht zu *wahr* auswertenden Klauseln, entweder nur positiv oder nur negativ auftritt, dann kann sie zur Vereinfachung der Formel so gesetzt werden, dass alle diese Klauseln zu *wahr* auswerten.

Unit-Propagation Wenn es eine *Unit-Klausel* gibt, so wird diese erfüllt und der Algorithmus fährt mit der nächsten Iteration fort. Eine Unit-Klausel ist eine Klausel, in der alle Literale, bis auf eines, bereits belegt sind und die selbst den Wahrheitswert *undefiniert* hat. Eine solche Klausel kann nur erfüllt werden, wenn das letzte nicht belegte Literal so belegt wird, dass die Klausel *wahr* wird.

Belegung einer freien Variable Eine freie Variable wird gewählt und auf den Wert *wahr* gesetzt.

Die Pure-Literal-Elimination wird nur für die Bonuspunkte notwendig sein. Der folgende Pseudocode beschreibt eine Umsetzung des Algorithmus:

```
setze alle Variablen auf UNDEFINIERT
while (true):
  if (alle Klauseln sind WAHR):
    return SAT
  else if (mindestens eine Klausel ist FALSCH):
    if (Rücksetzung möglich):
      Rücksetzung
    else:
      return UNSAT
  else if (mindestens eine Klausel ist UNDEFINIERT):
    if (Unit-Klausel (oder Pure-Literal) vorhanden):
      erfülle beliebige Unit-Klausel (oder beliebiges Pure-Literal)
    else: // nur wenn KEINE Unit-Klausel (und kein Pure-Literal) mehr vorhanden
      wähle nächste freie Variable x
      setze x auf WAHR
```

Weitere Details zum Algorithmus finden sich in dem Abschnitt auf Seite 5.

Projektaufbau

Klonen Sie das Projekt in einen beliebigen Ordner der virtuellen Maschine für die Vorlesung:

```
git clone https://prog2scm.cdl.uni-saarland.de/git/project2/$NAME /home/prog2/project2
```

wobei Sie $NAME durch Ihren CMS-Benutzernamen ersetzen müssen. Dazu benötigen Sie Ihr CMS-Passwort und müssen im Universitätsnetz bzw. dem Universitäts-VPN sein.
In dem Hauptverzeichnis Ihres Repos finden Sie jetzt die Verzeichnisse `src`, `include`, `test` und `inst` sowie die *Makefile*. In diesem Projekt geht es darum, dass Sie Ihr Programm von Grund auf selbst programmieren, daher sind keine `.c`- und `.h`-Dateien vorgegeben. Die Makefile ist jedoch vorkonfiguriert und sollte nicht bearbeitet werden, da bei der Evaluation die von uns mitgelieferte Version verwendet wird und Ihre sämtlichen Änderungen an dieser Datei verworfen werden. Benutzen Sie das Verzeichnis `include/` für Ihre `.h`-Dateien und das Verzeichnis `src/` für Ihre `.c`-Dateien. Die mitgelieferte Makefile wird bei der Kompilation genau die `.h`-Dateien, die sich in `include/` befinden und genau die `.c`-Dateien, die sich in `src/` befinden, berücksichtigen (ohne Sub-Verzeichnisse).
Sie können an dem Projekt mit einem Texteditor Ihrer Wahl, beispielsweise dem *Kate* Texteditor, der auf der virtuellen Maschine für die Vorlesung vorinstalliert ist, arbeiten. Das Projekt können Sie von der Befehlszeile aus übersetzen, indem Sie den Befehl

```
make
```

im Hauptverzeichnis Ihres Repos verwenden. Durch den Befehl

```
make check
```

können Sie die öffentlichen Tests ausführen. Dieser Befehl führt ein Skript aus, das Sie nach dem erfolgreichen Übersetzen des Projektes auch direkt ausführen können um einzelne Tests anhand Ihres Namens auszuführen:

```
test/run_tests.py -t "public.satsolver.rc_sat.tiny_sat"
```

führt beispielsweise den Test `public.satsolver.rc_sat.tiny_sat` aus.
Zum Untersuchen einzelner öffentlicher Tests können Sie zusätzlich mit dem `-v` Argument die vom Test ausgeführten Befehle anzeigen lassen.
Hinweis: Die Tests in Ihrem Repo werden zum Teil aus exact-3-CNF-Instanzen automatisch generiert: Wenn Sie selbst geschriebene exact-3-CNF-Instanzen (im unten beschriebenen Eingabeformat) lokal in das Verzeichnis `inst/public/auto/sat` bzw. `inst/public/auto/unsat` hinzufügen, werden dafür jeweils Tests generiert, die beim Ausführen des Testskripts mitausgeführt werden.

Programmspezifikation

Sowohl im Ein- als auch im Ausgabeformat werden Variablen durch echt-positive ganze Zahlen angegeben.[3] Für eine Variable x wird `x` notiert um ein positives Literal (im Eingabeformat) oder ein Belegung mit dem Wahrheitswert *wahr* (im Ausgabeformat) darzustellen. Um ein negatives Literal oder eine Belegung mit dem Wahrheitswert *falsch* darzustellen, wird `-x` geschrieben.

Eingabeformat

Eine Eingabedatei repräsentiert eine Formel in exact-3-CNF. Sie besteht aus zwei Teilen:

- der Kopfzeile (die erste Zeile der Datei)
- der Liste von Klauseln (alle folgenden Zeilen)

Die Kopfzeile fängt mit der Zeichenkette `p cnf` an, dann folgen getrennt durch je ein Leerzeichen die Zahlen n und m.[4] Dabei stellt n eine obere Schranke für die Anzahl der Variablen dar, und m gibt die exakte Anzahl an Klauseln an. Es sind sind keine weiteren Zeichen in der Kopfzeile.
Darauf folgen genau m Zeilen, die jeweils eine Klausel repräsentieren. Eine Klausel wird durch exakt 3 ganze Zahlen ungleich der Null codiert. Diese Zahlen sind durch je ein Leerzeichen getrennt und die Zeile wird durch eine `0` abgeschlossen. Für alle auftretenden Variablen x gilt $x \leq n$, es müssen aber nicht alle Variablen y mit $y \leq n$ in der Formel vorkommen.
Die Eingabedatei endet mit einer Leerzeile.
Die Beispieldatei

[3] Jede Variable passt in einen `int`-Behälter.
[4] n und m passen immer in einen `int`-Behälter.

```
p cnf 5 3
1 3 -2 0
3 -1 5 0
-2 -3 1 0
```

wird also als

$$(X_1 \vee X_3 \vee \neg X_2) \wedge (X_3 \vee \neg X_1 \vee X_5) \wedge (\neg X_2 \vee \neg X_3 \vee X_1) \tag{3}$$

interpretiert.

Ausgabeformat

Eine Ausgabedatei gibt an, ob eine Formel erfüllbar ist und gibt weiterhin wenn ja eine erfüllende Variablenbelegung an.
Wenn die Formel nicht erfüllbar ist, soll eine Datei ausgegeben werden, die ausschließlich aus der Zeichenfolge `UNSAT` und einem anschließendem Zeilenumbruch besteht.
Wenn die Formel erfüllbar ist, besteht die Ausgabedatei aus zwei Zeilen: In der ersten steht nur `SAT`. In der zweiten wird – aufsteigend, durch je ein Leerzeichen getrennt – jede Variable x mit $x \leq n$ (unabhängig davon ob x tatsächlich in der Formel vorkommt) in ihrer Polarität aufgelistet. Also `x` wenn x auf *wahr* gesetzt ist und $-x$, wenn x auf *falsch* gesetzt ist. Nach der letzten Variable folgen ein Leerzeichen, eine `0` und ein Zeilenumbruch. Es folgen keine weiteren Zeichen.
Eine Beispielausgabedatei mit einer erfüllenden Variablenbelegung für 3 ist:

```
SAT
1 2 -3 4 5 0
```

Kommandozeilenargumente

Ihr kompiliertes Programm wird mit dem Befehl

```
bin/satsolver <input-file> <output-file> [-p | -b]
```

ausgeführt. Das Programm nimmt als erstes Argument den Pfad zur Eingabedatei, welche in dem oben spezifizierten Eingabeformat sein sollte und als zweites Argument nimmt es einen gültigen Pfad für die Ausgabedatei. Die Flag `-b` ist nur für Bonuspunkte relevant und ihre Verwendung wird dort beschrieben. Die Flag `-p` unterscheidet zwei Operationsmodi, die mit den beiden Aufgaben des Projektes korrespondieren:

Finden von puren Literalen (7 Punkte). Wenn die Flag `-p` angegeben ist, soll Ihr Programm Dateien im angegebenen Eingabeformat einlesen und eine aufsteigende, mit je einem Leerzeichen getrennte Liste von allen *puren Literalen* in ihrer jeweiligen Polarität ausgeben. Pure Literale sind genau die Literale, die in der Formel nur in einer Polarität auftreten. Variablen die gar nicht auftreten aber kleiner gleich der Variablenschranke n sind, sollen in positiver Polarität in der Liste stehen. Darauf folgen ein Leerzeichen, eine `0` und ein Zeilenumbruch. Dahinter sollen keine weiteren Zeichen mehr stehen. Wenn die Beispieleingabedatei also als `inst/example.in` abgespeichert ist, dann sollte der Befehl

```
bin/satsolver inst/example.in inst/example.out -p
```

die Zeile

```
-2 4 5 0
```

in die Datei `inst/example.out` schreiben. Bei erfolgreicher Terminierung soll der Return Code `0` zurückgegeben werden.
Hinweis: Wenn Sie sich geschickt anstellen, können Sie die Suche nach puren Literalen in der Bonusaufgabe wiederverwenden.

SatSolver (13 Punkte). Wenn die Flag `-p` nicht angegeben ist, soll Ihr Programm die Eingabedatei einlesen und eine Ausgabedatei schreiben, welche im oben angegebenen Ausgabeformat angibt, ob die eingelesene exact-3-CNF Formel erfüllbar ist. Wenn die Formel erfüllbar ist soll das Programm eine vom DPLL-Algorithmus gefundene Variablenbelegung im korrekten Format angeben. Außerdem soll bei Terminierung Return Code `10` für SAT und `20` für UNSAT verwendet werden. Dabei ist es wichtig, dass die Optimierung Unit-Propagation vor jeder neuen Variablenbelegung, die nicht durch eine Optimierung impliziert wurde, ausgeführt wird, und zwar so oft wie möglich. Die Integration Tests werden mit einem Zeitlimit ausgeführt. Ein trivialen SatSolver, welcher alle Belegungen ausprobiert, wird bei den größeren Testinstanzen am Zeitlimit scheitern.

Pure-Literal-Elimination (2 Bonuspunkte). Wenn Sie die optionale Optimierung Pure-Literal-Elimination zusätzlich zu der Optimierung Unit-Propagation in ihren Algorithmus einbauen, können Sie Bonuspunkte erhalten. Genau wie bei der Unit-Propagation ist es hier wieder wichtig, dass diese Optimierung in jeder Iteration ausgeführt wird: Vor jeder neuen Variablenbelegung, die nicht durch eine der beiden Optimierungen impliziert wurde, muss Pure-Literal-Elimination durchgeführt werden. Die Tests für die Bonuspunkte werden immer mit der Flag `-b` ausgeführt. Dies bietet Ihnen die Möglichkeit, Pure-Literal-Elimination in ihrer normalen Ausführung (ohne `-b`) auszuschalten, da es unter Umständen ihr Programm auch verlangsamen kann.
Hinweis: In welcher Reihenfolge Unit-Propagation und Pure-Literal-Elimination in einer Iteration relativ zueinander ausgeführt werden, ist Ihnen überlassen.

Error-Handling Sie können davon ausgehen, dass ihr Programm nur mit validen Parametern ausgeführt wird, dass die Eingabedatei existiert, lesbar ist und ein valides Format hat und dass der Pfad für die Ausgabedatei valide und beschreibbar ist.

Implementierung

Im Folgenden werden einige kritische Punkte zur Implementierung des SatSolvers genauer erklärt. Solange Sie sich an die Programmspezifikation halten, können Sie von den folgenden Vorschlägen abweichen.
Hinweis: Implementieren Sie nicht alles auf einmal. Es empfiehlt sich meist *top-down* vorzugehen, also zuerst die grobe Struktur zu schreiben. Damit Ihr Programm immer kompiliert, können Sie die benutzten Methoden dann in passenden `.h`-Dateien spezifizieren und zunächst leere Implementierungen in die passenden `.c`-Dateien schreiben.

Main. Ihre `main.c` sollte den Parser aufrufen um die Eingabedatei einzulesen. Danach muss je nachdem ob die Flag `-p` mitgegeben wurde die richtige Subroutine aufgerufen werden. Mehr sollte nicht in Ihrer `main.c` stehen. Teilen Sie Ihren Code logisch in Dateien auf, das trägt sehr zur Übersichtlichkeit bei und hilft Ihnen beim Debuggen.

Datenstrukturen. Ein wichtiger Teil des Projektes ist, sich geeignete Datenstrukturen zu überlegen. Es bietet sich an eine Datenstruktur für die eingelesene exact-3-CNF-Formel und eine weitere für die aktuelle Variablenbelegung anzulegen.
Hinweis: Zum Einlesen können Sie die `C`-Standardbibliotheks-Funktion `fscanf` verwenden.

Implizierte und nicht-implizierte Variablenzuweisung. Während der Suche nach einer zulässigen Variablenbelegung ist es immer wieder notwendig, eine zuvor getroffene Entscheidung rückgängig zu machen, um nach einer anderen Belegung zu suchen. Dazu müssen Sie wissen, für welche Variablen es eine alternative Belegung gibt und für welche Variablen der einzig zulässig Wert inferiert wurde. Für jede Variablenbelegung muss also die Information vorhanden sein, ob ihr Wert ausgewählt wurde (im Folgenden als `CHOSEN` bezeichnet) oder ob ihr Wert inferiert wurde (im Folgenden als `IMPLIED` bezeichnet). Rückgesetzt können nur Werte werden, die `CHOSEN` sind, aber alle Variablenbelegungen, die daraus inferiert wurden – also alle `IMPLIED` Variablenbelegungen die *danach* hinzu kamen – müssen auch zurückgenommen werden. Neben der Information für jede Variable ob sie `CHOSEN` oder `IMPLIED` ist, ist also auch die Reihenfolge der Belegungen wichtig.

Stack

Es bietet sich an einen *Stack* (engl. für *Stapel*) zu benutzen um die Reihenfolge der Variablenbelegungen zu speichern. Ein Stack ist eine Datenstruktur, die folgende Operationen unterstützt:

- das Erstellen eines leeren Stacks (`stack_create`)
- das Hinzufügen eines Elements in den Stack (`stack_push`)
- das Abrufen des *neuesten* Elements (`stack_peek`) (falls vorhanden)
- das Löschen des *neuesten* Elements (`stack_pop`) (falls vorhanden)
- das Prüfen ob der Stack leer ist (`stack_isempty`)
- die Speicherfreigabe des Stacks (`stack_free`)

Man kann es sich also wirklich wie einen Stapel vorstellen: Ein neues Element wird oben aufgelegt, und auch nur auf dieses Element besteht direkter Zugriff.
Hinweis: Ein Stack ist eine spezielle Form der Liste, Sie können sich bei der Implementierung an der Liste im Skript orientieren.

Rücksetzung. Falls die Formel zu `FALSE` ausgewertet wird, wird die Rücksetzung (engl. *backtracking*) versucht: Es wird nach der zuletzt zugewiesenen Variable gesucht, die nicht durch eine Optimierung impliziert wurde (`CHOSEN`). Wenn eine `CHOSEN` Variable gefunden wird, dann wird der bisher zugehörige Wahrheitswert negiert und die Variable ist von nun an `IMPLIED` (die gegenteilige Belegung wurde ja bereits ausprobiert und hat zu einem Widerspruch geführt). Außerdem müssen `IMPLIED` Variablenbelegungen, die nach dieser Variable getätigt wurden zurückgenommen werden. Wenn es keine `CHOSEN` Variablenbelegung gibt, ist die Formel nicht erfüllbar.

Belegung einer freien Variable. In diesem Schritt raten Sie eine Variablenbelegung und mit jedem Raten wächst die Laufzeit exponentiell. Achten Sie daher unbedingt darauf, dass dies wirklich nur gemacht wird, wenn die implementierten Optimierungen keine weiteren Variablenbelegungen inferieren können!

Viel Erfolg!