Review

Last week

- Function Languages
- Lambda Calculus
- Scheme

Outline

- Programming Assignment 1
- Types

Sources:
Enderton, Herbert B. A Mathematical Introduction to Logic, Second Edition, chapter. 1
PLP, 7

Programming Assignment 1

Write a SAT solver in Scheme

A SAT solver solves the Boolean satisfiability problem.

In order to understand the satisfiability problem, we must first define the language in which the problem is phrased.

The language is propositional logic.
What is Logic?

Like a programming language, logic is defined by its syntax and semantics.

Syntax

- An alphabet is a set of symbols.
- A finite sequence of these symbols is called an expression.
- A set of rules defines the well-formed expressions.

Semantics

- Gives meaning to well-formed expressions
- Formal notions of induction and recursion can be used to provide a rigorous semantics.

Propositional Logic: Syntax

Alphabet

- ( Left parenthesis Begin group
- ) Right parenthesis End group
- ¬ Negation symbol English: not
- ∧ Conjunction symbol English: and
- ∨ Disjunction symbol English: or (inclusive)
- A\textsubscript{1} First propositional symbol
- A\textsubscript{2} Second propositional symbol
- . . . nth propositional symbol

Propositional Logic: Well-Formed Formulas

An expression is a sequence of symbols. We use an inductive definition to define the set \( W \) of well-formed formulas (wffs) in propositional logic.

- \( U \) = the set of all expressions.
- \( B \) = the set of expressions consisting of a single propositional symbol.
- \( F \) = the set of formula-building operations:
  - \( E_{¬}(\alpha) = (¬\alpha) \)
  - \( E_{∧}(\alpha, \beta) = (\alpha \land \beta) \)
  - \( E_{∨}(\alpha, \beta) = (\alpha \lor \beta) \)

\( W \) is the set generated from \( F \) by \( B \). In other words, it is the smallest set of expressions containing \( B \) and closed under \( F \).

Propositional Logic: Semantics

Intuitively, given a wff \( \alpha \) and a value (either \( T \) or \( F \)) for each propositional symbol in \( \alpha \), we should be able to determine the value of \( \alpha \).

Let \( v \) be a function from \( B \) to \{\( F \), \( T \)\}. We call this function a truth assignment.

Now, we define \( \overline{v} \), a function from \( W \) to \{\( F \), \( T \)\} as follows (we compute with \( F \) and \( T \) as if they were 0 and 1 respectively).

- For each propositional symbol \( A_i \), \( \overline{v}(A_i) = v(A_i) \).
- \( \overline{v}(E_{¬}(\alpha)) = T − \overline{v}(\alpha) \)
- \( \overline{v}(E_{∧}(\alpha, \beta)) = \min(\overline{v}(\alpha), \overline{v}(\beta)) \)
- \( \overline{v}(E_{∨}(\alpha, \beta)) = \max(\overline{v}(\alpha), \overline{v}(\beta)) \)
Truth Tables

There are other ways to present the semantics which are less formal but perhaps more intuitive.

<table>
<thead>
<tr>
<th>α</th>
<th>¬α</th>
<th>α ∧ β</th>
<th>α ∨ β</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Complex truth tables

Truth tables can also be used to calculate all possible values of \( \tau \) for a given wff. We associate a column with each propositional symbol and a column with each propositional connective. There is a row for each possible truth assignment to the propositional connectives.

<table>
<thead>
<tr>
<th>(A_1)</th>
<th>(A_2)</th>
<th>(A_3)</th>
<th>(\bar{A_1} \lor (A_2 \land \bar{A_3}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T T T F F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T T T T T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T T F F F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T F F T T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F F T F F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F T T F F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F F T T T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F F F F F</td>
</tr>
</tbody>
</table>

Definitions

If \(\alpha\) is a wff, then a truth assignment \(v\) satisfies \(\alpha\) if \(v(\alpha) = T\).

A wff \(\alpha\) is **satisfiable** if there exists some truth assignment \(v\) which satisfies \(\alpha\).

If there exists no truth assignment that satisfies \(\alpha\), then we say that \(\alpha\) is **unsatisfiable**.

If \(\alpha\) is true under every truth assignment, then we say \(\alpha\) is a **tautology** or \(\alpha\) is **valid**.

If \(\alpha\) and \(\beta\) have the same value under every truth assignment, then \(\alpha\) and \(\beta\) are **tautologically equivalent**.

Some tautological equivalences

**Associative and Commutative laws**

- \((A \land (B \land C)) = ((A \land B) \land C)\)
- \((A \lor (B \lor C)) = ((A \lor B) \lor C)\)
- \((A \land B) = (B \land A)\)
- \((A \lor B) = (B \lor A)\)

**Distributive Laws**

- \((A \land (B \lor C)) = ((A \land B) \lor (A \land C))\)
- \((A \lor (B \land C)) = ((A \lor B) \land (A \lor C))\)

**De Morgan’s Laws**

- \(\neg(A \land B) = (\neg A \lor \neg B)\)
- \(\neg(A \lor B) = (\neg A \land \neg B)\)
Determining Satisfiability using Truth Tables

An Algorithm for Satisfiability

To check whether \( \alpha \) is satisfiable, form the truth table for \( \alpha \). If there is a row in which \( T \) appears as the value for \( \alpha \), then \( \alpha \) is satisfiable. Otherwise, \( \alpha \) is unsatisfiable.

Conjunctive Normal Form

Most algorithms for determining satisfiability assume the input is in conjunctive normal form (CNF).

Some definitions:
- A literal is a propositional variable or its negation
- A clause is a disjunction of one or more literals
- A formula is in CNF if it consists of a conjunction of clauses
- A propositional symbol occurs positively if it occurs unnegated in a clause.
- A propositional symbol occurs negatively if it occurs negated in a clause.

Determining Satisfiability using Truth Tables

Example

\[ \alpha \land ((B \lor \neg A) \land (C \lor \neg B)) \]

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
<th>( \alpha \land ((B \lor \neg A) \land (C \lor \neg B)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Conjunctive Normal Form

Examples

- Literals: \( P_1, \neg P_1 \)
- Clauses: \( (P_1 \lor \neg P_3 \lor P_5), (P_2 \lor \neg P_2) \)
- CNF: \( (P_1 \lor \neg P_3) \land (\neg P_2 \lor P_3 \lor P_5) \)
- In the above formula, \( P_1 \) occurs positively and \( P_2 \) occurs negatively
Programming Assignment 1

Write a Scheme function called sat that takes a single argument, a list of integers representing a SAT problem in the DIMACS format. Your function should return #t if the formula is satisfiable and #f if the formula is not satisfiable.

The assignment will be due in three weeks (October 29).

Types

What is a type?

- A type consists of a set of values
- The compiler/interpreter defines a mapping of these values onto the underlying hardware.

CNF: Alternative notations

\[ \neg A \lor \neg B \lor E \land \neg E \lor A \land \neg E \lor B \land \neg C \lor F \land \neg F \lor C \land \neg D \lor \neg E \lor G \land \neg G \lor D \land \neg E \lor H \land \neg H \lor E \land G \lor H \lor I \land I \lor G \land I \lor H \land I \]

CNF: Alternative notations

**DIMACS standard**

Each variable is represented by a positive integer. A negative integer refers to the negation of the variable. Clauses are given as sequences of integers separated by spaces. A 0 terminates the clause.

\[
(A' + B' + E)(E' + A)(E' + B) \\
(C' + F)(F' + C) \\
(D' + E' + G)(G' + D)(G' + E) \\
(E' + F' + H)(H' + E)(H' + F) \\
(G + H + I')(I + G')(I + H') \\
(I)
\]

-1 -2 5 0 -5 1 0 -5 2 0 -3 6 0 -6 3 0 -4 -5 7 0 -7 4 0 -7 5 0 7 8 9 0 9 -7 0 9 -8 0 9 0
Types

What purpose do types serve in programming languages?

- Implicit context for operations
  - Makes programs easier to read and write
  - Example: \( a + b \) means different things depending on the types of \( a \) and \( b \).
- Constrain the set of correct programs
  - Type-checking catches many errors

Type Systems

A type system consists of:

- a mechanism for defining types and associating them with language constructs
- a set of rules for:
  - type equivalence: when do two objects have the same type?
  - type compatibility: where can objects of a given type be used?
  - type inference: how do you determine the type of an expression from the types of its parts

What constructs are types associated with?

- Constant values
- Names that can be bound to values
- Subroutines (sometimes)
- More complicated expressions built up from the above

Type Checking

Type checking is the process of ensuring that a program obeys the type system’s type compatibility rules.

A violation of the rules is called a type clash.

Languages differ in the way they implement type checking:

- strong vs weak
- static vs dynamic

Strong vs Weak Typing

- A strongly typed language does not allow variables to be used in a way inconsistent with their types (no loopholes)
- A weakly typed language allows many ways to bypass the type system (e.g., pointer arithmetic)

C is a poster child for the latter. Its motto is: “Trust the programmer”.
Static vs Dynamic Type Systems

Static vs Dynamic

- Static
  - Variables have types
  - Compiler ensures that type rules are obeyed at compile time
  - ADA, PASCAL, ML

- Dynamic
  - Variables do not have types, values do
  - Compiler ensures that type rules are obeyed at run time
  - LISP, SCHEME, SMALLTALK, scripting languages

A language may have a mixture: JAVA has a mostly static type system with some runtime checks.

Pros and cons

- static: faster (dynamic typing requires run-time checks), easier to understand and maintain code, better error checking
- dynamic: more flexible, easier to write code

Polymorphism

Polymorphism allows a single piece of code to work with objects of multiple types.

- Parametric polymorphism: types can be thought of as additional parameters
  - implicit: often used with dynamic typing: code is typeless, types checked at run-time (LISP, SCHEME) - can also be used with static typing (ML)
  - explicit: templates in C++, generics in JAVA

- Subtype polymorphism: the ability to treat a value of a subtype as a value of a supertype

- Class polymorphism: the ability to treat a class as one of its superclasses (special case of subtype polymorphism)

Parametric polymorphism example

SCHEME

```scheme
(define (length l)
  (cond
    ((null? l) 0)
    (#t (+ (length (cdr l)) 1))))
```

The types are checked at run-time.

ML

```ml
fun length xs =
  if null xs
  then 0
  else 1 + length (tl xs)
```

How can ML be statically typed and allow polymorphism?

It uses type variables for the unknown types. The type of this function is written `a list -> int.`

Assigning types

Programming languages support various methods for assigning types to program constructs:

- determined by syntax: the syntax of a variable determines its type (FORTRAN 77, ALGOL 60, BASIC)

- no compile-time bindings: dynamically typed languages

- explicit type declarations: most languages
Types: Points of View

Denotational
- type is a set $T$ of values
- value has type $T$ if it belongs to the set
- object has type $T$ if it is guaranteed to be bound to a value in $T$

Constructive
- type is either built-in (int, real, bool, char, etc.) or
- constructed using a type-constructor (record, array, set, etc.)

Abstraction-based
- Type is an interface consisting of a set of operations

Scalar Types Overview
- discrete types
  - must have clear successor, predecessor
  - integer types
    - often several sizes (e.g., 16 bit, 32 bit, 64 bit)
    - sometimes have signed and unsigned variants (e.g., C/C++, ADA, C#)
  - enumeration types
- floating-point types
  - typically 64 bit (double in C); sometimes 32 bit as well (float in C)
- rational types
  - used to represent exact fractions (SCHEME, LISP)
- complex
  - FORTRAN, SCHEME, LISP, C 99, C++ (in STL)

Other intrinsic types
- boolean
  - Common type; C had no boolean until C 99
- character, string
  - some languages have no character data type (e.g., JAVASCRIPT)
  - internationalization support
    - JAVA: UTF-16
    - C++: 8 or 16 bit characters; semantics implementation dependent
  - string mutability
    - most languages allow it, JAVA does not.
- void, unit
  - Used as return type of procedures;
  - void (C, JAVA) represents the absence of a type
  - unit: (ML, HASKELL) a type with one value: ()

Enumeration types: abstraction at its best
- trivial and compact implementation:
  - values are mapped to successive integers
- very common abstraction: list of names, properties
- expressive of real-world domain, hides machine representation

Examples:
- type Suit is (Hearts, Diamonds, Spades, Clubs);
- type Direction is (East, West, North, South);

Order of list means that Spades > Hearts, etc.

Contrast this with C#:

“arithmetics on enum numbers may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error”
### Enumeration types and strong typing

**type** Fruit is (Apple, Orange, Grape, Apricot);
**type** Vendor is (Apple, IBM, HP, Dell);

My_PC : Vendor;
Dessert : Fruit;
...
My_PC := Apple;
Dessert := Apple;
Dessert := My_PC; -- error

Apple is **overloaded**. It can be of type Fruit or Vendor.

Overloading is allowed in C#, JAVA, ADA

Not allowed in PASCAL, C

### Subranges

ADA and PASCAL allow types to be defined which are subranges of existing discrete types.

**type** Sub is new Positive range 2 .. 5; -- Ada
V : Sub;

**type** sub = 2 .. 5; (* Pascal *)
var v : sub;

Assignments to these variables are checked at runtime:

V := I + J; -- runtime error if not in range

### Composite Types

- arrays
- records
- variant records, unions
- pointers, references
- function types
- lists
- sets
- maps

### Composite Literals

Does the language support these?

- array aggregates
  
  A := (1, 2, 3, 10); -- positional
  A := (1, others => 0); -- for default
  A := (1..3 => 1, 4 => -999); -- named

- record aggregates
  
  R := (name => "NYU", zipcode => 10012);
Type checking and inference

- Type checking:
  - Variables are declared with their type.
  - Compiler determines if variables are used in accordance with their type declarations.

- Type inference: (ML, Haskell)
  - Variables are declared, but not their type.
  - Compiler determines type of a variable from its usage/initialization.

In both cases, type inconsistencies are reported at compile time.

```fun f x =
    if x = 5 (* There are two type errors here *)
    then hd x
    else tl x
```

Type equivalence

Name vs structural

- name equivalence
  two types are the same only if they have the same name (each type definition introduces a new type)
- strict: aliases (i.e. declaring a type to be equal to another type) are distinct
- loose: aliases are equivalent

- structural equivalence
  two types are equivalent if they have the same structure

Most languages have mixture, e.g., C: name equivalence for records (structs), structural equivalence for almost everything else.

Type equivalence example

Structural equivalence in ML:

```type t1 = { a: int, b: real };

type t2 = { b: real, a: int };

(* t1 and t2 are equivalent types *)
```

Accidental structural equivalence

```type student = {
    name: string,
    address: string
}

type school = {
    name: string,
    address: string
}

type age = float;

type weight = float;
```

With structural equivalence, we can accidentally assign a `school` to a `student`, or an `age` to a `weight`.
Type conversion

Sometimes, we want to convert between types:

- if types are structurally equivalent, conversion is trivial (even if language uses name equivalence)
- if types are different, but share a representation, conversion requires no run-time code
- if types are represented differently, conversion may require run-time code (from `int` to `float` in C)

A **nonconverting type cast** changes the type without running any conversion code. These are dangerous but sometimes necessary in low-level code:

- `unchecked_conversion` in ADA
- `reinterpret_cast` in C++