Progetto di Programmazione 2018
`SharpSolver`

v2.3

December 21, 2018

Abstract

In questo documento vengono spiegate del dettaglio le specifiche del progetto `SharpSolver` per l'esame di Introduzione alla Programmazione a.a. 2018-19. Questo documento offre sia un ripasso degli argomenti che vi serviranno per svolgere il progetto, sia un approfondimento di alcuni aspetti avanzati di F#, utili allo svolgimento del progetto. Consigliamo di **controllare regolarmente** la **versione** e la **data** di aggiornamento di questo documento in quanto sarà soggetto a revisioni future al fine di rendere più chiare le specifiche del progetto .

Indice

1 Introduzione

Il progetto consiste nell'implementazione di un programma in grado di manipolare espressioni polinomiali, in particolare di **semplificare**, **normalizzare** e **derivare** polinomi e **risolvere equazioni** polinomiali fino al secondo grado. Vi viene richiesto di completare l'implementazione di un programma che è già impostato: viene fornito uno scheletro di codice che è *già* in grado di leggere da terminale i comandi e le espressioni polinomiali inserite dell'utente con un apposito *parser* e costruire le strutture dati adeguate. Il vostro compito è quello di **completare l'implementazione** andando a modificare i file **Main.fs** e **Impl.fs** secondo le istruzioni fornite in questo documento.

2 Indicazioni generali

- Si accettano gruppi di massimo **3** persone.
- Leggere con attenzione le istruzioni di consegna che trovate nella home del corso, progetti **consegnati in modo errato** verranno considerati **insufficienti**.
- Progetti **non compilanti** verranno considerati **insufficienti**.
- **Non** è consentito cambiare la struttura del progetto, è obbligatorio che il progetto compili nella sua interezza, potete modificare i file **Main.fs** e **Impl.fs**.

3 Plagio

I progetti verranno controllati al fine di verificare che non vi sia stato plagio tra consegne di gruppi differenti. Non vogliamo ostacolare il normale scambio di idee tra gruppi e l'aiuto reciproco ma nel momento in cui vi mettete a scrivere il vostro codice dovete farlo in **totale indipendenza** dagli altri gruppi. In caso di plagio si considera **annullato l'esame di laboratorio** per tutti i componenti dei gruppi coinvolti, indipendentemente da chi abbia copiato il progetto di chi, **tutti gli studenti coinvolti sono ritenuti ugualmente responsabili del plagio**.

3.1 Sul serio: non copiate

In modo meno formale e più personale vi consigliamo vivamente di non copiare, cercate di essere sicuri che il vostro progetto non possa essere ricondotto a quello di altri gruppi, vi assicuro che se succede abbiamo una precisione del 99% nel riconoscere il plagio dalla semplice casualità. Se scopriamo un plagio siamo costretti a intraprendere una serie di azioni che dispiacciono

tanto a voi quanto a noi (credetemi non lo dico per semplice retorica), il professore di laboratorio può decidere di annullarvi l'esame e l'universita può bloccare la vostra carriera accademica per un anno intero. Se avete problemi nello svolgimento del progetto venite a chiedere aiuto al professore, ai tutor o agli assistenti, non ci facciamo problemi ad aiutarvi se dimostrate interesse e impegno ed è più sicuro che copiare codice scritto da altri che non sareste in grado di spiegare all'orale (inoltre è gratis e siamo tutti molto simpatici).

4 Elementi di valutazione

Sebbene il progetto sia un lavoro che può essere svolto in gruppo, la **valutazione** sarà **individuale**. Oltre alla discussione orale, saranno valutate:

- qualità del **codice**: oltre alla correttezza, saranno considerate l'eleganza della soluzione implementata, l'utilizzo di indentazione corretta e consistente, la suddivisione in sotto-funzioni, etc;
- qualità della **documentazione**: codice **non commentato** non sarà valutato;

5 Indicazioni per la consegna

Il progetto è contenuto nella cartella `SharpSolver` contenuta nell'omonimo zip. Vi viene richiesto di consegnare un file `SharpSolver.zip` contenente la cartella `SharpSolver` che a sua volta contiene il vostro progetto: essenzialmente dovete consegnare lo stesso zip che scaricate con i file `Main.fs` e `Impl.fs` modificati da voi con le vostre implementazioni.

6 Concetti di matematica

Il progetto di quest'anno è fortemente inspirato a concetti di matematica: polinomi, equazioni, derivate ed altri fondamenti dell'algebra. In questa sezione cercheremo di presentarvi velocemente alcune nozioni fondamentali legate ai polinomi. I concetti presentati dovrebbero essere già stati trattati esaustivamente nei corsi di Calcolo e Matematica del primo anno, per questo motivo in questa sezione proponiamo un veloce ripasso dei concetti strettamente necessari allo svolgimento del progetto. Nel caso in cui le spiegazioni riportate non fossero sufficienti vi consigliamo di consultare i **riferimenti bibliografici** che trovate alla fine di questo documento.

- In matematica un **monomio** è un'espressione algebrica costituita da un coefficiente ed una parte letterale dove tra le lettere compaiono moltiplicazioni e elevamenti a potenza aventi esponente naturale;

- In matematica un **polinomio** [1] è un'espressione composta da costanti e variabili combinate usando soltanto addizione, sottrazione e moltiplicazione. In altre parole, un polinomio *tipico*, cioè *ridotto in forma normale*, è la somma algebrica di **monomi** non simili tra loro, vale a dire con parti letterali e grado diversi;

In questo progetto tratteremo sempre e solo **polinomi con 1 sola variabile** x. Tali polinomi possono essere visti come **funzioni** in 1 variabile - ad esempio, il seguente polinomio può essere rappresentato da una funzione $P(x)$:

$$P(x) \quad = \quad 7x^2 + 5x - 3$$

L'applicazione di un argomento alla funzione $P(x)$ produrrebbe la sostituzione di tutte le occorrenze di x nell'espressione del polinomio a destra del simbolo di uguale (=). Quale significato hanno dunque le **radici**? Le radici di un polinomio sono quei valori che, sostituiti alle variabili, riducono l'espressione polinomiale a **0**. Il che equivale a risolvere la seguente equazione:

$$7x^2 + 5x - 3 \quad = \quad 0$$

Formalmente l'insieme delle soluzioni (o radici del polinomio) è definito come $S = \{\ x \in \mathbb{R} \mid P(x) = 0\ \}$.

Per **polinomio in forma normale** (anche detto *polinomio normalizzato*) si intende un polinomio formato da **monomi** che hanno tutti parte letterale e grado diversi e compaiono nel polinomio in ordine **decrescente** di grado. Nel nostro caso, trattando polinomi in una sola variabile, la cosa si riduce alla diversità del grado.

Mostriamo ad esempio un polinomio *non* normalizzato di grado 3:

$$2x + 4x^2 + 3 - x + x^3 + 6x^2 - 1\,.$$

La normalizzazione consiste nel raccogliere i coefficienti di tutti i termini **simili** (cioè aventi parte letterale di grado uguale) e sommarli:

$$(2-1)x + (4+6)x^2 + x^3 + 3 - 1\,.$$

Riducendo le somme e riordinando i monomi in ordine descrescente per grado otteniamo un polinomio normalizzato:

$$x^3 + 10x^2 + x + 2\,.$$

6.1 Equazioni di grado 0 o *identità*

Possiamo intendere come equazione di grado 0 una equazione in cui non compare nessuna incognita, cioè una uguaglianza tra due espressioni riducibili a

costanti numeriche. Risolvere una equazione di grado 0 significa di fatto *verificare una identità*, pertanto si tratta di una operazione avente un risultato booleano - una identità può essere vera o falsa, ad esempio:

$$\begin{aligned} 5+3 &= 7 \\ 5+3-7 &= 0 \\ -2 &= 0 \end{aligned}$$

Dopo la normalizzazione, spostando il polinomio di destra sul lato sinistro monomio per monomio[1], l'equazione si riduce ad una costante confrontata con zero. L'identità $c = 0$ è verificata se c è effettivamente uguale a 0; falsa altrimenti.

6.2 Equazioni di 1^o grado

Le equazioni di **primo grado** [2] in una incognita sono equazioni in cui la x è elevata a esponente 1 e può essere solamente moltiplicata o sommata a costanti numeriche o espressioni comunque costanti - espressioni in cui, in altre parole, non compare la x. Un'equazione di primo grado è un'uguaglianza tra due polinomi che, una volta ridotti in forma normale e raccolti da un solo lato, riducono l'equazione ad una uguaglianza tra un polinomio e 0.

$$\begin{aligned} 4x+5 &= 2x \\ 4x+5-2x &= 0 \\ 2x+5 &= 0 \\ x &= -\tfrac{5}{2}\,. \end{aligned}$$

La **normalizzazione di una equazione**, dunque, è una operazione che da 2 polinomi $P_1(x)$ e $P_2(x)$ produce 1 solo polinomio $P'(x)$, il quale è la normalizzazione della differenza di polinomi $P_1(x) - P_2(x)$. Trovare le soluzioni dell'equazione significa trovare le radici di $P'(x)$. La procedura è rappresentata dai seguenti passaggi:

$$\begin{aligned} P_1(x) &= P_2(x) && \texttt{forma iniziale} \\ P_1(x) - P_2(x) &= 0 && \texttt{spostiamo tutto a sinistra} \\ P'(x) &= 0 && \texttt{equazione normalizzata} \end{aligned}$$

Per trovare le soluzioni dell'equazione, ovvero le radici di $P'(x)$, è necessario isolare la variabile x tramite manipolazione di termini:

$$bx + c = 0$$

$$x = -\frac{c}{b}\,.$$

[1] Sebbene appaiano come costanti numeriche, dal punto di vista sintattico sono monomi di grado 0.

6.3 Equazioni di 2^o grado

Un'equazione di **secondo grado** [3][4] è un'equazione polinomiale in cui l'incognita compare almeno una volta con esponente di grado 2 e tale che l'incognita compaia con grado massimo pari a 2.

$$4x^2 + 5 = 2x\,.$$

La forma normale di una equazione di secondo grado è definita come segue:

$$ax^2 + bx + c = 0\,.$$

La risoluzione delle equazioni di secondo grado richiede il calcolo del discriminante:

$$\Delta = b^2 - 4ac\,.$$

Il segno del discriminante ci permette di capire la molteplicità e il tipo delle soluzioni dell'equazione di secondo grado:

- se il discriminante ha valore **positivo**, l'equazione ha due soluzioni reali distinte.
- se il discriminante è **0**, l'equazione ha due soluzioni reali coincidenti.
- se il discriminante ha valore **negativo**, l'equazione non ammette soluzioni reali.

Il discriminante ci permette di scrivere la formula risolutiva:

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}\,.$$

6.4 Derivazione di un polinomio

In matematica, la **derivata** [5] è la misura di quanto la crescita di una funzione cambi al variare del suo argomento. L'interpretazione *geometrica* di una derivata è la pendenza della tangente al grafico di una funzione in un dato punto. Nel caso di una funzione $f(x)$ a una variabile reale, la derivata $f'(x)$ è la funzione dei coefficienti angolari m delle rette[2] tangenti ad ogni punto della primitiva $f(x)$.

Dal punto di vista *algebrico* è possibile calcolare la derivata di una funzione in maniera simbolica, unicamente tramite manipolazione di termini. Come accennato in sezione 6, un polinomio può essere interpretato come una funzione in 1 variabile x, sulla quale deriviamo applicando alla lettera le regole di derivazione dei polinomi [6].

[2]Dove la funzione di una retta ha forma $y = mx + q$.

In generale possiamo intendere la **derivazione di un monomio** come la *riscrittura* di un monomio, ovvero come una operazione che mappa monomi in monomi $D : \mathbb{M} \to \mathbb{M}$, dove $\mathbb{M}$ è l'insieme dei monomi di grado 0 o superiore in 1 variabile x.

$$\begin{array}{rcll} D[cx^n] & = & (n \cdot c)x^{n-1} & \texttt{grado } n >= 1 \\ D[c] & = & 0 & \texttt{costante} \end{array}$$

Ricordiamo che le costanti sono monomi di grado 0, cioè $D[c] \equiv D[cx^0]$.

Questa operazione può essere estesa alla **derivazione di polinomi**. La derivata di un polinomio diventa una operazione $D : \mathbb{P} \to \mathbb{P}$ che mappa polinomi in polinomi, dove $\mathbb{P} \supset \mathbb{M}$ è l'insieme dei polinomi in 1 variabile x di grado 0 o superiore. La seguente relazione mostra come, per derivare un polinomio, sia sufficiente sommare le derivate dei monomi che lo costituiscono:

$$D\left[\sum_{i=0}^{n} c_i x^i\right] = \sum_{i=0}^{n} D[c_i x^i]$$

Ad esempio:

$$D[2x^2 + 3x + 5] = 2(2x^{2-1}) + 1(3x^{1-1}) + 0 = 4x + 3$$

Altro esempio:

$$D[2x^4 + 3x^3 - 5x^2] = 4(2x^{4-1}) + 3(3x^{3-1}) + 2(-5x^{2-1}) = 8x^3 + 9x^2 - 10x\,.$$

Va notato che la derivata di un polinomio di grado n è un polinomio di grado $n - 1$, dove il *lower bound* $n >= 0$ garantisce che la derivata di una costante (cioè di un monomio di grado 0) sia 0 e non un monomio di grado negativo[3]

7 Specifica del progetto

Il progetto consiste nell'implementare un programma in grado di manipolare polinomi e compiere alcune operazioni su di essi. `SharpSolver` si comporta come un interprete di comandi, o una console interattiva, in cui l'utente può inserire da tastiera espressioni polinomiali che rispettano una certa sintassi ed il programma scrive in output il risultato.

[3]Virtualmente, applicando la regola generale, derivare un monomio di grado 0 dovrebbe produrre un monomio di grado -1, ad esempio $D[3] \equiv D[3x^0] = (-1 \cdot 3)x^{-1} = -\frac{3}{x}$ - ma non è così.

7.1 Moduli

Viene fornito un progetto F# per Visual Studio 2017 con lo scheletro del programma ed i tipi di dato principali già definiti. Il codice è suddiviso in diversi file sorgente, ciascuno dei quali definisce un **modulo** - una raccolta mista di definizioni di funzioni e/o tipi. Ogni sorgente dichiara in alto il nome del modulo, ad esempio la dichiarazione `module SharpSolver.Absyn` in testa al sorgente `Absyn.fs` definisce il modulo `Absyn` all'interno del *namespace* `SharpSolver`, che è il nome radice di tutto il progetto. Moduli e namespace permettono di organizzare il codice in parti e sottoparti innestate ad albero, in maniera del tutto simile ai percorsi di file e cartelle in un filesystem.

Segue ora una descrizione di ciascun modulo e di ciò che contiene:

Prelude Raccoglie tipi e funzioni globali utili a tutto il programma. La definizione del tipo `rational` è qui.

Config Contiene definizioni di costanti globali utili all'intero programma, come stringhe di testo e valori numerici di configurazione.

Absyn Definisce i tipi che rappresentano la **sintassi astratta** (***Ab****stract* ***Syn****tax* in inglese) le entità sintattiche che il programma manipola: i tipi `monomial` e `polynomial` sono definiti qui come tipi *heavyweight*, cioè tipi unione aventi un solo costruttore - si veda la sezione 8.1.1 per maggiori dettagli.

Parser Il file `Parser.fs` con il modulo `Parser` è generato automaticamente da un programma esterno che Visual Studio chiama automaticamente durante la compilazione. Si tratta di `fsyacc`, il porting per F# del celebre generatore di parser *Yacc*. Il file `Parser.fsy` funge da input per `fsyacc`, specificando la *grammatica* che il parser dovrà rispettare e le regole per costruire le strutture dati definite in `Absyn.fsy` opportunamente. La sintassi in notazione BNF è riportata in forma semplificata nella tabella 1[4].

Lexer Il lexer è generato anch'esso da un programma esterno invocato durante la compilazione - `fslex`, il generatore di lexer di F#. Similmente al parser, nel file `Lexer.fsl` compare la specifica delle regole lessicali che il programma deve riconoscere. In pratica si occupa di riconoscere le keyword, le parentesi, i caratteri speciali come gli operatori e le costanti numeriche, trasformando l'input da una stringa a una sequenza di *token*, cioè una sequenza di simboli terminali che poi il parser è in grado di analizzare secondo le regole grammaticali specificate.

[4]Per approfondire le grammatiche formali si consulti [7], per la notazione EBNF [8] e per il parsing in generale [9]. In breve: i simboli non-terminali sono in maiuscolo e quelli terminali sono sottolineati; coefficienti re;

Table 1: Sintassi astratta del linguaggio riconosciuto da `SharpSolver` .

L	$\rightarrow$		**linea**
		$\underline{\texttt{\#}}\text{s}$	comando
	$\mid$	E	espressione
	$\mid$	$E \ \underline{=} \ E$	equazione
E	$\rightarrow$		**espressioni**
		P	polinomio
	$\mid$	$\underline{\texttt{D[}E\texttt{]}}$	derivata
P	$\rightarrow$		**polinomi**
		M	monomio
	$\mid$	$P \ \underline{+} \ M$	somme di monomi
M	$\rightarrow$		**monomi**
		c	costante
	$\mid$	$c\underline{\texttt{x}}^n$	grado

dove s è una stringa, $c \in \mathbb{Q}$, $n \in \mathbb{N}$ con $n \geqslant 1$.

Impl Il modulo `Impl` è definito in due file: `Impl.fsi` contiene le firme delle funzioni, mentre il file `Impl.fs` contiene le implementazioni vere e proprie. **Questo è il modulo che raccoglie la gran parte del codice da scrivere per questo progetto**. Si badi che per aggiungere nuove funzioni *pubbliche*[5], a questo modulo è necessario dichiarare la loro firma nel file `.fsi`, altrimenti non saranno visibili dagli altri moduli - lo scopo del file `.fsi` è proprio filtrare le funzioni definite nel file `.fs` , rendendo visibili dall'esterno (cioè da altri moduli del programma) solamente quelle di cui viene dichiarata la firma nel file `.fsi`.

Main contiene la funzione *main* e le funzioni principali che si occupano dell'interprete di comandi. All'interno ci sono svariati commenti e `TODO`, ovvero punti del sorgente in cui è necessario intervenire aggiungendo codice. **Anche questo modulo richiede l'aggiunta di codice**: in particolare, nel costrutto `match` dentro il corpo della funzione `interpreter_loop` è necessario aggiungere i pattern per gestire valori di tipo `line` mancanti, al fine di rendere il pattern matching esaustivo.

L'ordine dei file sorgente in un progetto F# conta: `Main.fs` è l'ultimo perché deve poter vedere tutte le definizioni (di tipi e/o funzioni) degli altri moduli; in generale ogni sorgente deve stare *sotto* i sorgenti di cui utilizza i nomi (di funzioni e/o di tipi), esattamente allo stesso modo in cui, all'interno di un singolo sorgente F#, ogni let-binding deve stare sotto i let-binding a cui fa riferimento.

Aggiungere nuove funzioni pubbliche naturalmente è possibile: introducendo ad esempio una nuova funzione nel modulo `Absyn.fs`, sarà possibile utilizzarla da tutte le funzioni definite sotto nel modulo stesso ed anche da qualunque altro modulo posizionato sotto nell'ordine dei file di progetto in Visual Studio.

7.2 Rational

Rappresentiamo i numeri razionali (l'insieme $\mathbb{Q}$ in matematica) tramite un apposito tipo `rational` definito nel modulo `Prelude`. Un numero razionale è di fatto una frazione tra interi: lo rappresentiamo internamente come una coppia di interi (p, q) dove p è il numeratore e q è il denominatore. Tale rappresentazione - ripetiamo - è **interna**, cioè non visibile dall'esterno: questo significa che un valore di tipo `rational` non è visto come una coppia, ma come un tipo a sé stante, distinto dalla coppia - in maniera simile al modo in cui un valore di tipo `monomial` non è una coppia ma conserva al suo interno una coppia, come spiegato in sezione 7.1. Tuttavia, al suo interno, sono conservati il numeratore ed il denominatore sotto forma di una coppia.

[5] Vengono chiamati *pubblici* i let-binding globali a *top-level* di un modulo

E' possibile costruire un valore di tipo `rational` applicando il costruttore `rational` (il costruttore si chiama come il tipo, in questo caso) ad una coppia di interi, rispettivamente numeratore e denominatore; internamente la frazione viene sempre semplificata. E' anche possibile scrivere *literal* di tipo `rational` aggiungendo il suffisso `Q` ad una costante intera positiva - si badi che i literal sono *sempre e solo* costanti. Diamo alcuni esempi chiarificatori per la costruzione di numeri razionali:

```
let q1 = 5Q                  // 5/1
let q2 = -9Q                 // -9/1
let q3 = rational (5, 1)     // 5/1 come q1
let q4 = rational 5          // 5/1 come q3 e q1
let q5 = rational (11, 8)    // 11/8
let q6 = rational (-11, 8)   // -11/8
let q7 = rational (11, -8)   // -11/8 come q6
let q8 = rational (50, 15)   // semplificato a 10/3
```

Tra le operazioni supportate dal tipo `rational` ci sono gli operatori di somma `(+)`, sottrazione `(-)`, moltiplicazione `(*)`, divisione `(/)` e negazione unaria `(~-)`, oltre a svarianti altri metodi[6] come la radice quadrata `Sqrt`, il valore assoluto `Abs` e la potenza `Pow`. Il tipo `rational` supporta anche il confronto e l'uguaglianza.

Per conoscere il valore del numeratore o del denominatore di un `rational` è necessario invocare i metodi `N` e `D`. Ad esempio, se volessimo definire una funzione che inverte un numero razionale e restituisce il suo *reciproco*:

```
let invert_rational (q : rational) = rational (q.D, q.N)
```

Un altro esempio - una funzione che calcola il quadrato di un numero razionale può essere scritta in due modi:

```
let square_rational (q : rational) = q ** 2

let square_rational' (q : rational) =
    rational (int (float q.N ** 2.0),
              int (float q.D ** 2.0))
```

La prima versione utilizza l'operatore binario `(**)` che invoca automaticamente il metodo `Pow` di `rational`; tale metodo vuole un `int` come esponente perché è definito così in `Absyn.sf`. La seconda versione invece estrae numeratore e denominatore usando i metodi `N` e `D`, li converte a `float`, calcola il quadrato di entrambi ed infine li riconverte ad `int` per passarli in coppia al costruttore di `rational`. E' interessante notare che l'operatore `(**)` è usato con i `float`, in questo caso, e pertanto l'esponente `2.0` è anch'esso un `float` - questo perché l'operatore `(**)` per i `float` è pre-definito in F# e vuole un `float` anche come esponente.

Si consulti il codice nel modulo `Absyn` per ulteriori dettagli implementativi.

[6] Si chiamano *metodi* le funzioni definite come *membri* di un tipo.

8 Tipi di dato e AST

La nostra rappresentazione di monomi e polinomi è **sintattica**, non numerica. Questo significa che le strutture dati che modellano le entità in gioco non sono semplici valori numerici, ma tipi di dato strutturati, appositamente definiti per catturare le informazioni di natura sintattica associate ad esse.

Un esempio approssimativo ma chiarificatore: ipotizzando come input una espressione `7 - (8 - 2)`, la rappresentazione in memoria non è valore `13`, che sarebbe il valore a cui l'espressione può essere ridotta, ma una lista `[ 7; -8; 2 ]` che rappresenta l'espressione sotto forma di sequenza di costanti additive, senza parentesi. Il parser - fornito con lo scheletro di codice - è in grado di analizzare la stringa di testo inserita dall'utente in input e creare in memoria un **albero di sintassi astratta**[7] che riflette la struttura sintattica dell'input. Inoltre, il parser si occupa anche di sbrigare alcune operazioni preliminari di semplificazione, come togliere le parentesi e ridistribuire la meno, nel nostro caso.

8.1 Monomial e polynomial

Per rappresentare un monomio cx^n è necessario conservare 2 informazioni: il coefficiente c ed il grado n; la variabile non fa parte della rappresentazione di un monomio perché, per semplicità, non permettiamo all'utente di usare variabili diverse da x direttamente a livello di sintassi. Ipoteticamente, se avessimo permesso all'utente di inserire polinomi con nomi di variabili qualunque, il nome della variabile sarebbe stata un'informazione importante da conservare nella struttura dati che rappresenta un monomio, che dovrebbe diventare una tripla `(coefficiente, grado, variabile): rational * int * string`.

8.1.1 Tipi *heavyweight* vs tipi *lightweight*

Possiamo, in sintesi, rappresentare un monomio come una coppia (c, n) dove $c \in \mathbb{Q}$ e $n \in \mathbb{N}$. Ma anziché definire i monomi come un *type alias* in F#:

```
type monomial = rational * int
```

abbiamo definito un *nuovo tipo di dato* con un costruttore `Monomial`:

```
type monomial = Monomial of rational * int
```

La differenza è sottile: la prima definizione non introduce un nuovo tipo ma solamente un *alias* di nome `monomial` per il tipo coppia `rational * int`. I *type alias* sono anche detti tipi **lightweight**, cioè *leggeri*. Un parallelo con il mondo dei valori può essere utile:

[7] Viene chiamata *Abstract Syntax Tree* (AST) la struttura dati che rappresenta una informazione sintattica, anche quando non è propriamente un albero [10].

```
let one = 1
let f x = x + one
```

Nell'esempio, abbiamo introdotto un nuovo binding tra il nome `one` ed il valore `1` di tipo `int`. Il simbolo `one` è utilizzabile al posto di `1`, come accade nel corpo della funzione `f`, poiché è equivalente in tutto e per tutto a scrivere `1`. Non è un nuovo valore, è solamente un nome per un valore che esiste già: definire un *type alias* è un po' come fare un `let`, ma nel mondo dei tipi.

Ciò è profondamente diverso da definire un *nuovo* tipo; un nuovo tipo introduce anche dei nuovi valori o costruttori, allo stesso modo in cui per definire un nuovo insieme è necessario definire anche gli elementi che lo compongono. Quando si definisce un *tipo unione*[8] in F# si introduce un nuovo tipo con nuovi valori; se il tipo ha un solo costruttore, tuttavia, non è propriamente un tipo unione e viene chiamato un tipo **heavyweight**, cioè *pesante*.

Tornando ai monomi, per costruire un valore di tipo `monomial` è necessario applicare il costruttore `Monomial` ad una coppia di tipo `rational * int`, alla stessa maniera in cui si applica una funzione ad un argomento. Immaginiamo di voler costruire il monomio $3x^2$: il primo `let` lo fa definendo una coppia mentre il secondo costruisce un valore di tipo `monomial`:

```
let m1 = (3Q, 2)           // m1 : rational * int
let m2 = Monomial (3Q, 2) // m2 : monomial
let m3 = Monomial m1       // m3 : monomial
```

Solo `m2` ed `m3` hanno tipo `monomial`.

8.1.2 Pattern matching sul `let`

L'operazione inversa alla costruzione - cioè il pattern matching - è particolarmente facile per i tipi *heavyweight* perché hanno un solo costruttore. Ad esempio, immaginiamo di voler implementare una funzione `monomial_negate : monomial -> monomial` che dato un monomio ne calcola la negazione aritmetica negando il segno del coefficiente e lasciando il grado inalterato:

```
let monomial_negate m =
    match m with
    | Monomial (coeff, deg) -> Monomial (-coeff, deg)
```

Il pattern matching è necessario per decostruire un valore di tipo `monomial`: senza pattern matching non potremmo dare un nome al primo ed al secondo elemento della coppia *agganciata* al costruttore `Monomial` - e senza dare nomi alle cose, non possiamo usarle. Il risultato in output è calcolato costruendo un nuovo valore di tipo `monomial` con il coefficiente negato e lo stesso grado

[8] I tipi unione vengono anche chiamati *disjoint unions*, *discriminated unions*, *algebraic datatypes* oppure *variant* in letteratura o da altri linguaggi di programmazione [11].

del monomio in input: nessuna modifica dei dati dunque, ma solamente decostruzione e ricostruzione.

Per evitare la verbosità del costrutto `match` con un solo pattern, si può rendere più conciso il codice scrivendo il pattern direttamente sul parametro della funzione. Tecnicamente, il costrutto `let` di F# è in grado di fare pattern matching tanto quanto il costrutto `match`, ma con una limitazione: non è possibile specificare casi di pattern multipli separati dal pipe (`|`). Non è indicato per i tipi unione con più costruttori dunque, poiché il compilatore produrrebbe un errore di *pattern matching non esaustivo*, ma quando si tratta di tipi *heavyweight* è molto comodo:

```
let monomial_negate (Monomial (coeff, deg)) =
    Monomial (-coeff, deg)
```

Analogamente, per costruire un valore di tipo `polynomial` è necessario applicare il costruttore `Polynomial` alla lista di `monomial` che esso si aspetta come argomento:

```
let p1 = Polynomial [ Monomial (1Q, 2); Monomial (2Q, 1);
    Monomial (3Q, 3) ]
let monos = [ Monomial (3Q, 5); Monomial (2Q, 1); ]
let p2 = Polynomial monos
```

Le liste di `monomial` tra parentesi quadrate non sono `polynomial` di per sé: `monos` ha tipo `monomial list` e lo stesso tipo è inferito anche per la sotto-espressione tra parentesi quadrate nel primo binding. Ad una lista di `monomial` bisogna applicare il costruttore `Polynomial` per costrure un `polynomial`, esattamente allo stesso modo in cui una coppia `rational * int` non è un `monomial` di per sé, ma è necessario applicare il costruttore `Monomial`.

Attenzione: **i costruttori *non* sono tipi**. Si noti la differenza tra il tipo `monomial` ed il costruttore `Monomial`: il primo è un **tipo**, non un valore; il secondo invece è un **costruttore**, cioè un **valore** che si comporta come una funzione, permettendo l'applicazione ad un argomento (la coppia `rational * int` in questo caso). Il motivo per cui vengono chiamati *costruttori* è precisamente questo: sono funzioni che *costruiscono* un valore di un tipo nuovo a partire da un input che funge da argomento. A ulteriore conferma di questo, si faccia caso al tipo che dà F# al costruttore:

$$\texttt{Monomial} : \texttt{rational * int} \rightarrow \texttt{monomial}$$

Questo è un tipo con la freccia, il che conferma che il costruttore `Monomial` è visto come una funzione da F#; una funzione che, appunto, prende come argomento una coppia di tipo `rational * int` e produce un valore di tipo `monomial`.

8.2 Polinomi normalizzati

Il tipo `normalized_polynomial` rappresenta un polinomio normalizzato - vedi sezione 6 per il significato matematico. E' un tipo totalmente distinto dal tipo `polynomial`: per F# non c'è relazione tra i due, sebbene in matematica sappiamo che l'insieme dei polinomi normalizzati è un sottoinsieme dell'insieme dei polinomi.

Il tipo `normalized_polynomial` è un tipo *heavyweight* con un solo costruttore `NormalizedPolynomial` che si aspetta un argomento di tipo `rational[]`, cioè un **array con i coefficienti razionali ordinati per grado** in ordine crescente[9].

La ragione per cui rappresentiamo un polinomio normalizzato con un array di coefficienti anziché con una lista di monomi è sottile: una lista di valori di tipo `monomial` può essere liberamente popolata con monomi qualunque, non c'è modo di imporre una disciplina forte sul contenuto della lista; in un array, invece, l'indice funge implicitamente da grado e l'elemento contenuto è il relativo coefficiente.

In generale, infatti, possiamo interpretare un array di τ come una mappa che associa interi positivi a valori di tipo τ, dove l'indice intero funge da chiave.

8.2.1 Expr e line

Nel modulo `Absyn` sono definiti altri due tipi che rappresentano entità sintattiche del linguaggio che `SharpSolver` interpreta. Il tipo `expr` rappresenta le espressioni polinomiali. La differenza tra un polinomio ed una espressione polinomiale è che una espressione polinomiale può essere 2 cose, ciascuna rappresentata da un costruttore:

Poly rappresenta un `polinomial` vero e proprio;

Derive rappresenta la derivata di una `expr`, ricorsivamente.

Questo tipo è ricorsivo perché la sintassi è ricorsiva: infatti è possibile innestare derivate a piacere, ad esempio una espressione polinomiale `D[D[5x + 1]]` produce `Derive (Derive (Poly (Polynomial [ .. ])))`, mentre una espressione polinomiale semplice, senza derivate, come `5x^3 + 3x - 1]]` produce `Poly (Polynomial [ .. ])`. Il costruttore `Poly` funge da caso base e si applica ad un `polynomial`.

Per chiarezza mostriamo la differenza tra i vari tipi con dei semplici esempi di binding:

[9] In F# il tipo parametrico `array` può essere usato al posto delle parentesi quadrate dopo il nome del tipo. In altre parole, `T[]` ≡ `T array` per ogni tipo `T`.

```
let m1 = Monomial (1Q, 3)                 // m1 : monomial
let m2 = Monomial (rational (7, 5), 1)    // m2 : monomial
let p = Polynomial [m1; m2]               // p : polynomial
let e = Poly p                            // e : expr
```

Il tipo `line` rappresenta le 3 possibili forme che può assumere l'input inserito dall'utente, ciascuna rappresentata da un costruttore:

Expr rappresenta una **espressione polinomiale**;

Equ rappresenta **equazione tra due espressioni polinomiali**

Cmd rappresenta un comando che comincia col carattere `'#'` (hash).

Nella tabella di seguito riportiamo alcuni esempi di input e relativo AST costruito dal parser, al fine di chiarire i meccanismi di innestamento dei dati ed il ruolo dei vari costruttori.

Input	**AST**
`#help`	`Cmd "help"`
`#foobar`	`Cmd "foobar"`
`7x + 1`	`Expr (Poly (Polynomial [Monomial (7Q, 1);` `Monomial (1Q, 0)]))`
`D[D[7x + 1]]`	`Expr (Derive (Derive (Poly (Polynomial [..]))))`
`6x + 1 = 8 - 3x2`	`Equ (Poly (Polynomial [..]),` `Poly (Polynomial [..]))`
`D[D[6x + 1] = D[8 - 3x2]`	`Equ (Derive (Derive (Poly (Polynomial [..]))),` `Derive (Poly (Polynomial [..])))`

Per capire a fondo come il parser costruisce le strutture dati, si sperimenti con `SharpSolver` inserendo espressioni polinomiali diverse, anche con derivate, e si badi alla riga di log `absyn` - essa è il pretty-printing dell'AST.

8.3 Cosa implementare

Abbiamo già accennato in sezione 7.1 che il modulo `Main` contiene già lo scheletro di un pattern matching sul valore di tipo `line` prodotto dal parser. I pattern per i comandi `#help` e `#exit` sono già implementati. Completare il programma significa aggiungere i casi *mancanti* al pattern matching, cioè i casi dei costruttori `Expr` ed `Equ`, con la opportuna implementazione:

```
let line = Parser.line Lexer.tokenize lexbuf
match line with
| Cmd "help" -> out "%s" Config.help_text
```

```
| Cmd ("quit" | "exit") -> out "%s" Config.exit_text;
      exit 0

| Expr e1 ->
    // deriva l'espressione polinomiale e1
    // poi normalizza il polinomio risultate

| Equ (e1, e2) ->
    // deriva le espressioni polinomiali e1 ed e2
    // poi unisci lato destro e sinistro
    // e produci un unico polinomio normalizzato
    // infine trovane le radici con algoritmi diversi
    // a seconda del grado
```

Il codice che scriverete chiamerà svariate funzioni, poiché non vogliamo implementare tutto dentro il costrutto `match`. Tali funzioni sono raccolte nel file `Impl.fs`: i nomi ed i tipi di queste funzioni danno già molti indizi sulla logica degli algoritmi e fungono da guida per l'implementazione del programma. Molte di queste funzioni, in particolare quelle più complesse, probabilmente saranno chiamate da quelle più semplici: è una prassi assolutamente normale programmare spezzettando il lavoro in piccole funzioni che chiamano altre piccole funzioni, che a loro volta ne chiamano altre. Ad esempio, la funzione `solve2`, che trova le radici di un polinomio di secondo grado normalizzato, probabilmente avrà bisogno di conosce il grado dei monomi o del polinomio stesso. Il riuso il codice è una pratica elegante e desiderabile: anziché reimplementare gli stessi piccoli algoritmi più volte, definite piccole funzioni e chiamatele.

```
val rationalize : float -> rational

val monomial_degree : monomial -> int
val monomial_negate : monomial -> monomial
val polynomial_degree : polynomial -> int
val polynomial_negate : polynomial -> polynomial
val normalized_polynomial_degree :
   normalized_polynomial -> int

val normalize : polynomial -> normalized_polynomial
val reduce : expr -> polynomial
val derive : polynomial -> polynomial

val solve0 : normalized_polynomial -> bool
val solve1 : normalized_polynomial -> rational
val solve2 : normalized_polynomial -> (float * float
   option) option
```

Forniamo ora i dettagli di ciascuna di queste funzioni.

```
val rationalize : float -> rational
```

Prende in input un numero float e restituisce la sua rappresentazione in numero razionale. Ad esempio, il float 4.5 può essere razionalizzato come 45/10. Questa funzione è usata dal parser per convertire in un `rational` i coefficienti con la virgola inseriti dall'utente. Il motivo per cui è necessario razionalizzare i float è che un `monomial`, secondo la nostra definizione, supporta solamente coefficienti razionali, non numeri reali qualunque. Qualunque float può essere trasformato in una frazione con al denominatore una opportuna potenza di 10.

Ci sono due approcci possibili per implementare questa funzione: l'approccio aritmetico è quello più ovvio, ma esiste anche un secondo approccio basato su un trucco con le stringhe, di cui suggeriamo l'algoritmo:

1. stampare il float in una stringa tramite la funzione `sprintf` (simile a `printf` ma invece di stampare a video produce una stringa);
2. fare uno *split* della stringa usando il punto (il carattere `'.'`) come separatore usando il metodo `Split` - ad esempio, se `s` è il nome della stringa, l'espressione `s.Split ['.']` produce un array di 2 stringhe con la parte a sinistra del punto (la parte intera) e la parte a destra del punto (la parte decimale);
3. unire la stringa con la parte sinistra a quella con la parte destra creando una nuova stringa con l'intero numero senza virgole - si usi l'operatore `+` tra stringhe per concatenarle;
4. convertire la stringa appena concatenata in un numero di tipo `int` usando la funzione `Int32.Parse` - l'intero ottenuto è il *numeratore* del numero razionale da restituire in output;
5. per calcolare il *denominatore* si calcoli la potenza di 10 elevato alla lunghezza della parte decimale della stringa splittata precedentemente.

```
val monomial_degree : monomial -> int
```

Restituisce il grado del monomio dato in input.

```
val monomial_negate : monomial -> monomial
```

Prende in input un monomio e restituisce la sua negazione algebrica.

```
val polynomial_degree : polynomial -> int
```

Prende in input un polinomio e restituisce il suo grado. Ricordiamo che il grado di un polinomio corrisponde al grado del monomio di grado più alto.

```
val polynomial_negate : polynomial -> polynomial
```

Prende in input un polinomio e restituisce il polinomio negato, cioè il polinomio con tutti i monomi negati.

```
val normalized_polynomial_degree : normalized_polynomial
    -> int
```

Prende in input un polinomio normalizzato e restituisce il suo grado.

```
val normalize : polynomial -> normalized_polynomial
```

Prende in input un polinomio e restituisce il suo corrispondente polinomio normalizzato. Implementare questa funzione richiede la creazione di un array, che può essere affrontata in più modi:

- creando una lista con i coefficienti ordinati opportunamente e convertendola all'ultimo in un array tramite la funzione di libreria `List.toArray` - questo approccio è consigliato a coloro che vogliono scrivere un algoritmo ricorsivo sulle liste per manipolare i monomi e poi convertire facilmente in un array il risultato finale.
- costruendo un array tramite le funzioni del modulo `Array` di F#, per esempio `Array.init` o `Array.create`, e poi popolandolo opportunamente - questo approccio è consigliato a coloro che desiderano cimentarsi con la programmazione imperativa, creando un array e poi impostando i suoi elementi tramite assegnamenti in un ciclo iterativo;
- usando le *computation expression* di F# per creare direttamente l'array computando i suoi elementi - quest'ultimo approccio è il più sofisticato ed è raccomandato solamente per coloro che desiderano approfondire le feature più avanzate di F# [12].

```
val reduce : expr -> polynomial
```

Prende in input un'espressione e la semplifica fino a restituire il polinomio da essa rappresentato. Come spiegato in sezione 8.2.1, un valore di tipo `expr` rappresenta una espressione polinomiale, che può essere solo 2 cose: un polinomio oppure la derivata di una espressione polinomiale. Questo produce una struttura dati ricorsiva, come detto, che può essere convertita in un polinomio applicando ricorsivamente la funzione `derive`.

```
val derive : polynomial -> polynomial
```

Prende in input un polinomio e restituisce la sua derivata. Questa funzione trasforma polinomi in polinomi, calcolando la derivata secondo la procedura discussa in sezione 6.4.

```
val solve0 : normalized_polynomial -> bool
```

Questa funzione prende in input un polinomio e risolve l'equazione di grado zero che si ottiene eguagliando il polinomio dato a zero.

```
val solve1 : normalized_polynomial -> rational
```

Questa funzione prende in input un polinomio e risolve l'equazione di primo grado che si ottiene eguagliando il polinomio dato a zero.

```
val solve2 : normalized_polynomial -> (float * float
    option) option
```

Questa funzione prende in input un polinomio e risolve l'equazione di secondo grado che si ottiene eguagliando il polinomio dato a zero. La funzione può restituire zero, una o due soluzioni in base al numero di soluzioni:

`None` significa che l'equazione di secondo grado non ha soluzioni;

`Some (x1, None)` significa che l'equazione ha solamente 1 soluzione x_1;

`Some (x1, Some x2)` significa che l'equazione ha 2 soluzioni: x_1 e x_2.

8.4 Note sull'output

L'ouptut del programma è formato da varie righe di *log* che presentano i seguenti *tag*:

- [**absyn**]: stampa l'AST che il programma è riuscito a costruire a partire dalla stringa fornita in input;
- [**pretty**]: stampa l'AST con una notazione più leggibile e simile alla matematica;
- [**redux**]: mostra l'espressione ottenuta dopo l'applicazione della funzione `reduce`, nel caso in cui non sia richiesto il calcolo di derivate risulta essere uguale alla linea `[pretty]`;
- [**norm**]: mostra il risultato della normalizzazione;
- [**degree**]: mostra il grado del polinomio o dell'equazione *dopo* la normalizzazione;
- [**sol**]: viene mostrato solamente in caso l'input sia una equazione e ne mostra le soluzioni;
- [**ident**]: viene mostrato solamente in caso l'input sia una identità e mostra se è vera oppure no.

Il codice che aggiungerete al modulo `Main` dovrà utilizzare le funzioni di log definite all'inizio del sorgente `Main.fs` stesso per stampare in output le righe di log con i risultati. Ad esempio, si chiami la funzione `norm` passandogli il polinomio normalizzato per stamparlo in output; oppure la funzione `redux` per stampare il polinomio derivato restituito dalla `reduce`; oppure ancora la funzione `sol` per stampare le soluzioni di una equazione.

Proponiamo ora alcuni esempi di output.

8.5 Esempio 1

```
>> 5x + 3
[absyn]     Expr (Poly (Polynomial [Monomial (5,1);
   Monomial (3,0)]))
[pretty]    5x + 3
[redux]     5x + 3
[norm]      3 + 5x
[degree]    1
```

8.6 Esempio 2

```
>> 5x + 3 = 0
[absyn]     Equ
  (Poly (Polynomial [Monomial (5,1); Monomial (3,0)]),
   Poly (Polynomial [Monomial (0,0)]))
[pretty]    5x + 3 = 0
[redux]     5x + 3 = 0
[norm]      3 + 5x = 0
[degree]    1
[sol]       x = -3/5
```

8.7 Esempio 3

```
>> 5x^2 + 4x - 3
[absyn]     Expr (Poly (Polynomial [Monomial (5,2);
   Monomial (4,1); Monomial (-3,0)]))
[pretty]    5x^2 + 4x - 3
[redux]     5x^2 + 4x - 3
[norm]      -3 + 4x + 5x^2
[degree]    2
```

8.8 Esempio 4

```
>> 5x^2 + 4x - 3 = 0
[absyn]     Equ
  (Poly (Polynomial [Monomial (5,2); Monomial (4,1);
     Monomial (-3,0)]),
   Poly (Polynomial [Monomial (0,0)]))
[pretty]    5x^2 + 4x - 3 = 0
[redux]     5x^2 + 4x - 3 = 0
[norm]      -3 + 4x + 5x^2 = 0
[degree]    2
[sol]       x1 = 0.471779788708135 vel x2 =
   -1.27177978870813
```

8.9 Esempio 5

```
>> D[5x^2 + 4x - 3] = D[3x]
[absyn]     Equ
```

```
  (Derive (Poly (Polynomial [Monomial (5,2); Monomial
     (4,1); Monomial (-3,0)])),
   Derive (Poly (Polynomial [Monomial (3,1)])))
[pretty]    D[5x^2 + 4x - 3] = D[3x]
[redux]     10x + 4 = 3
[norm]      1 + 10x = 0
[degree]    1
[sol]       x = -1/10
```

8.10 Esempio 6

```
>> D[D[5x^2 + 4x - 3]] = D[D[3x]]
[absyn]     Equ
  (Derive
     (Derive
        (Poly (Polynomial [Monomial (5,2); Monomial (4,1)
           ; Monomial (-3,0)]))),
   Derive (Derive (Poly (Polynomial [Monomial (3,1)]))))
[pretty]    D[D[5x^2 + 4x - 3]] = D[D[3x]]
[redux]     10 = 0
[norm]      10 = 0
[degree]    0
[identity]  false
```

9 Uso della libreria standard di F#

Al fine di non complicare inutilmente l'implementazione del progetto vi viene consentito - anzi, consigliato - utilizzare le funzioni di libreria offerte dal linguaggio. Di seguito ne elenchiamo alcune che potrebbero tornarvi utili:

- List.map
- List.filter
- List.sortByDescending
- List.maxBy
- List.groupBy
- List.tryPick
- List.sumBy
- Array.sub
- Array.findIndexBack
- String.length

- System.printf
- System.sprintf
- Int32.Parse

Per maggiori informazioni vi consigliamo di leggere la documentazione ufficiale dei moduli `List` [13], `Array` [14] e `String` [15] di F#. In generale, cercando con Google il nome di un modulo o di una funzione F# è facile trovare documentazione.

10 Progetto *Avanzato* (opzionale)

Per gli studenti che desiderano cimentarsi in qualcosa di più sfidante, proponiamo una feature avanzata da implementare in `SharpSolver` : la risoluzione di **equazioni di terzo grado**. Anziché rigettare l'input dell'utente quando consiste in una equazione di grado superiore al secondo, si implementi l'algoritmo di risoluzione delle equazioni di terzo grado [16] [17].

References

[1] Polinomio. [Online]. Available: https://it.wikipedia.org/wiki/Polinomio

[2] Equazioni di primo grado. [Online]. Available: https://www.youmath.it/lezioni/algebra-elementare/equazioni/44-equazioni-di-i-grado-ad-una-incognita-prima-parte.html

[3] Equazioni di secondo grado. [Online]. Available: https://www.youmath.it/lezioni/algebra-elementare/equazioni/68-equazioni-di-secondo-grado-ad-unincognita-come-si-risolvono.html

[4] Equazioni di secondo grado. [Online]. Available: https://it.wikipedia.org/wiki/Equazione_di_secondo_grado

[5] Derivata. [Online]. Available: https://it.wikipedia.org/wiki/Derivata

[6] "Derivatives of polynomials," [Apri link].

[7] "Formal grammar," [Apri link].

[8] "Extended backus–naur form," [Apri link].

[9] "Parsing," [Apri link].

[10] "Abstract syntax tree," [Apri link].

[11] "Union types," [Apri link].

[12] "Computation expressions in f#," [Apri link].

[13] Collections.list module. [Online]. Available: https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/collections.list-module-%5bfsharp%5d

[14] Collections.array module. [Online]. Available: https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/collections.array-module-%5Bfsharp%5D

[15] System.string module. [Online]. Available: https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/core.string-module-%5Bfsharp%5D?f=255&MSPPError=-2147217396

[16] "Lezioni di matematica: Equazioni di grado superiore al secondo," [Apri link].

[17] "Equazioni di terzo grado," [Apri link].