Exercise `02_elispy`

Dr. Günter Kolousek

10. Dezember 2018

Dieses Beispiel soll einen Lexer, einen Parser, einen Interpreter und auch einen Transpiler (Compiler) für eine Untermenge der Programmiersprache LISP implementieren.

1 Allgemeine Instruktionen

- Just follow the instructions of the previous excercise…
- In addition to the already well-known facts, the purpose of this and the other excercises is not in the stupid and boring copying of program text or stupidly following instructions. It's clear as daylight, you are a 4th year student and it's **your** own responsibility to train your programming skills!

2 Überblick

- Die Idee ist einen Lexer, einen Parser, einen Interpreter und einen Transpiler (übersetzt Sourcecode in Sourcecode) für eine Lisp-artige Programmiersprache zu schreiben. Wir wollen dieser netten kleinen Programmiersprache den Namen `elispy` (klingt niedlich und ist angelehnt an die Emacs-Lisp-Variante elisp) geben.
- Die Sprache soll folgende Atome kennen:
  - ganze Zahlen, wie z.B. `-1`, `0`, oder `42`
  - Strings, wie z.B. `"abc"`

  - Symbole, wie z.B. `i` oder `count`
    * sowie die vordefinierten Symbole `t` und `nil`

- Folgende Funktionen sollen eingebaut sein: TBD
  - `+`, `-`, `*`, `/`
  - `<`, `<=`, `=`, `>`, `>=`
  - `first`, `rest`, `cons`, `equal`
  - `setq`
  - `if`
  - `while`
  - `shell`

3 Prinzipielle Funktionsweise

Dieser Abschnitt ist zu *lesen*, d.h. das Programmieren muss bis zum nächsten Abschnitt warten!

> Geduld ist bitter, aber ihre Früchte sind süß.
>
> – Aristoteles

3.1 Lexer

Ein Lexer funktioniert so, dass dieser den Quelltext (prinzipiell zeichenweise) liest und aus diesem eine Folge von Tokens (Zeichen, die eine bestimmte Bedeutung aufweisen) generiert. Ein Token enthält

- den Typ des Tokens
- den Wert des Tokens
- und die Position im Eingabestrom

Ein Lexem (engl. lexeme) ist eine konkrete Zeichenfolge, die ein Token ausmacht.

An Tokentypen gibt es meist:

- Identifier
- Keyword
- Operator
- Delimiter
- Literal

In der Regel wird der Lexer Whitespace-Zeichen und Kommentare *überlesen*. Unserer zumindest wird dies so tun, da Whitespace-Zeichen keinerlei Relevanz in Lisp haben (→ Klammern…)!

Weiters ist es die Aufgabe des Lexers, lexikalische Fehler zu erkennen und diese zuverlässig zu melden.

Betrachten wir die folgenden `C++`-Anweisungen:

```
int i;
cout << (1 + 2.1415926);
```

Hier finden sich die folgenden Lexeme: `int` (Keyword), `i` (Identifier), `;` (Delimiter), `cout` (Identifier), `<<` (Operator), `(` (Delimiter), `1` (Literal), `+` (Operator), `2.1415926` (Literal), `)` (Delimiter) und `;` (Delimiter)

Lisp ist hier bedeutend einfacher, denn es gibt keine Keywords und die Syntax ist auf Grund der sexps auch sehr simpel, wobei anstatt eines Identifier der Begriff Symbol verwendet wird:

```
(setq a (quote (1 (+ 2 3))))
```

Delimiter, Symbol, Symbol, Delimiter, Symbol, Delimiter, Literal, Delimiter, Symbol (!),…

Als syntaktische Abkürzung zur obigen sexp kann man bekanntlich auch schreiben:

```
(setq a '(1 (+ 2 3)))
```

Es gibt unterschiedliche Möglichkeiten wie so ein Lexer implementiert werden kann:

1. manuelles zeichenweises Lesen und Implementieren eines endlichen Automaten (mühsam)
2. verwenden von regulären Ausdrücken (einfacher, aber fragiler, da reguläre Ausdrücke…)

Wir verwenden die zweite Variante, eh klar…

Der prinzipielle Ablauf funktioniert so:

1. Alle "regulären Ausdrücke" in einer Liste speichern (Reihenfolge ist in der Regel *wichtig*!)

2. Einen regulären Ausdruck auf den Eingabestring anwenden

3. Wenn eine Übereinstimmung gefunden, dann ein Token erzeugen und es dem Benutzer zur Verfügung stellen. Handelt es sich um ein Token, das ignoriert werden soll (wie zum Leerzeichen), dann wird es eben *nicht* dem Benutzer zur Verfügung gestellt. Du siehst, dass es auch Tokens geben kann, die wir prinzipiell ignorieren wollen. Dies werden wir in weiterer Folge in der Implementierung auch so umsetzen.

4. Weiter mit Punkt 2 bis keine weiteren regulären Ausdrücke vorhanden.

5. Wenn kein einziger reguläre Ausdruck gepasst, hat dann liegt ein Fehler vor.

6. Ansonsten weiter mit Punkt 2) und wieder mit dem ersten regulären Ausdruck beginnen und diesen auf den Rest des Strings anwenden!

7. Fertig ist man wenn kein Fehler aufgetreten ist und der ganze String abgearbeitet wurde.

3.2 Parser

Der Parser selbst ist an sich kein großes Problem, wenn man diesen nicht optimieren und auch nicht verändern will, da man diesen für unsere gewählte Programmiersprache `elispy` leicht als *recursive-descent* Parser implementieren kann (siehe Folien über Compilertechnologie).

3.3 Interpreter

Der Interpreter soll als REPL (read eval print loop) in der Art des Python-REPL ausgeführt werden.

```
>>> (+ 1 2)
3
>>>
```

D.h. dem Benutzer soll ein TUI (Text User Interface) eben wie in Python oder einer Shell angeboten werden, sodass dieser eine sexp nach dem Prompt (`>>>`) eingeben kann und der Interpreter diese auswertet und den Wert der sexp in der nächsten Zeile ausgegeben wird.

3.4 Transpiler

Ein "transpiler" ist ein source-to-source compiler (kurz: transcompiler oder eben transpiler), der Sourcecode in Sourcecode übersetzt. Das ist unter Umständen kein optimaler Ansatz (wg. Performance und einem Zwischenschritt), aber es erleichtert unsere Aufgabe ungemein. Nebenbei gesagt war die allererste Version von `C++` (C with Classes) ebenfalls auf diese Art und Weise implementiert. Es handelte sich (klarerweise) um eine Umsetzung in die Programmiersprache C. Dieser Transpiler wurde von Bjarne Stroustroup `cfront` genannt.

Codegenerierung. TBD

4 Nun zum Programmieren!

Für dieses Beispiel (und auch die folgenden) gilt, dass die Klassen sich in einem Namespace befinden müssen, der auf deine Matrikelnummer lautet. Prinzipiell soll sich jede Klasse in einer eigenen Datei befinden, auch wenn dies in C# nicht unbedingt notwendig ist.

4.1 Hauptprogramm und Lexer

1. Das Programm soll sich in einer Datei `Elispy.cs` befinden und u.a. das Parsen der Kommandozeile beinhalten.

   Die Hilfeausgabe soll folgendermaßen aussehen und beschreibt auch die Kommandozeilenschnittstelle (in groben Zügen):

   ```
   usage: elispy [--help|-h|-g] [FILE]
   Executes the "elispy" expressions contained in FILE otherwise the REPL will be start

     --help|-h ... Help!
     -g ... generate C# code; only valid if FILE is provided
     FILE ... file name or - (stdin). If FILE is missing start the REPL
   ```

- Wenn `-h` oder `--help` als Kommandozeilenargument angegeben wird, dann wird die Hilfe ausgegeben und das Programm wird beendet.

- Wird die Option `-g` angegeben, dann soll (in weiterer Folge) das erzeugte C# Programm in einer Datei abgespeichert werden. Der Namen dieser Datei wird so gebildet, dass bei `FILE` (wenn es sich nicht um `-` handelt) eine etwaige Erweiterung (extension) durch die Erweiterung `.cs` ersetzt wird. Wird kein Kommandozeilenargument `FILE` mitgegeben, dann soll sich das Programm mit einer Fehlermeldung (siehe "usage") beenden.

- Wird `FILE` nicht angegeben, dann soll das Programm die Funktion eines REPL erfüllen. D.h. zuerst wird ein Prompt ausgegeben, danach wird sexps eingelesen, ausgewertet und danach in einer neuen Zeile der ermittelte Wert geschrieben. Die Realisierung wird im Abschnitt Interpreter beschrieben.

- Wird `FILE` angegeben

  – und hat den Wert `-`, dann bedeutet dies, dass von `stdin` gelesen wird und die Ausgaben auf `stdout` ausgegeben werden. Im Falle, dass `-g` angegeben worden ist, soll die Ausgabe des generierten C# Programmes in die Datei `Program.cs` geschrieben werden.

  – Wird für `FILE` ein "richtiger" Dateinamen angegeben, dann soll von dieser Datei gelesen werden und die Ausgaben auf `stdout` ausgegeben werden. Im Falle, dass `-g` angegeben wird, greift der Mechanismus wie bei der Option `-g` beschrieben.

  Dies wird im Abschnitt Transpiler beschrieben.

Die Abarbeitung der Optionen und Parameter, der Start des REPL,… ist natürlich zu diesem Zeitpunkt *nicht* zu implementieren! Hier geht es lediglich darum, dass die Verarbeitung der Kommandozeilenargumente richtig abgearbeitet wird (Funktionen `main`, `usage`, `parse_argv` und auch eine geeignete Struktur zur Aufnahme der entsprechenden Daten).

Damit ich es nicht vergesse: Auch diese Kommandozeilenverarbeitung ist (natürlich) wieder mit einem endlichen Automaten zu implementieren!

2. Jetzt zum Lexer!

   Alle Klassen, die mit dem Lexer in Verbindung stehen, kommen in den Namensraum `<matnr>.lexer`.

   a) Implementiere jetzt den Lexer…

      Hier ein paar Anweisungen und auch Tipps:

- Tokens werden durch die folgenden Klassen beschrieben:
  - Klasse `Definition`, die angibt wie ein spezielles Token aufgebaut ist. Dazu benötigt es:
    * einen Typ (`string`), der angibt um welche Art von Token es sich handelt, also z.B.
      · `LPAREN` für eine linke Klammer,
      · `RPAREN` für eine rechte Klammer,
      · `SYMBOL` für ein Symbol,
      · `INTEGER` für eine ganze Zahl,
      · `STRING` für ein Stringliteral,
      · `SPACE` für Whitespace-Zeichen und
      · `QUOTE` für das Terminalzeichen `'`.

      Dieser Typ wird uns als eindeutige ID für die Definition dienen (dafür ist aber nichts zu programmieren).

      Beachte wie wir die Tokens "entworfen" (festgelegt) haben. Natürlich könnte dies auch auf eine andere Weise realisiert worden sein, aber… Der Grund liegt darin, dass wir den Parser dann einfacher realisieren und auf die Grammatik zu optimieren können. Besonders das `QUOTE` Literal ist in diesem Zusammenhang interessant, da es ja eigentlich nicht wirklich in der Sprache der sexps vorkommt, sondern es sich um eine Abkürzung der Verwendung des Symbols `quote` handelt. LISP Interpreter können dies auf verschiedene Arten implementieren, wie z.B. mit Hilfe eines Macros (aber das ist eine andere Geschichte…).
    * einen regulären Ausdruck als Instanz der Klasse `Regex`, der angibt wie ein Lexem auszusehen hat, dem das Token zugeordnet ist. Siehe dazu die Erklärungen dazu weiter unten im Dokument!
    * und einen boolschen Wert, der angibt, ob das Token ignoriert werden soll. Das ist praktisch, wenn man Sprachen parsen will, bei denen gewissen Token zwar erkannt werden sollen, wie z.B. Whitespace-Zeichen, aber in weiterer Folge ignoriert werden. Das ist in unserer Sprache elispy so, in Python aber nicht…

Jede dieser drei Angaben ist durch ein Property zu implementieren. Da keine spezielle Verarbeitung notwendig ist, kann dieses als "auto-implemented" Property implementiert werden, d.h. einfach mit `get; set;` im Rumpf des Property. Setzen soll eines dieser Properties soll von außerhalb der Klasse nicht möglich sein!

Der Konstruktor soll folgende Signatur haben:

```
public Definition (string type, string regex, bool is_ignored)
```

D.h. im Konstruktor soll eine entsprechende Instanz der Klasse `Regex` angelegt werden. Im Konstruktor kannst du dafür einfach

```
this.regex = new Regex(regex, RegexOptions.Compiled);
```

schreiben. Weiters ist zu beachten, dass die Klasse `Regex` im Namensraum `System.Text.RegularExpressions` zu finden ist.

So, jetzt hast du alle Informationen, um die Klasse `Definition` implementieren zu können. Los geht's!

- Struktur `Position`, die einen Index im String, die Zeilennummer und die Spaltennummer enthält. Die letzten beiden Angaben sind notwendig, um eine vernünftige Fehlermeldung liefern zu können.

  Und wiederum ist ein Konstruktor zu schreiben und der Zugriff auf die Attribute über Properties zu realisieren.

  Eine `ToString` Methode ist zu implementieren, sodass die Position auf der Konsole ausgegeben werden kann.

- Struktur `Token`, die den Typ (einen String, der eindeutig den Typ der `Definition` angibt), den aktuellen Wert (also das Lexem als String) und die Position (eine Instanz von `Position`) enthält. Diese drei Angaben sind auch einem zu implementierenden Konstruktur als Parameter hinzuzufügen und auch dafür sind Properties zu verwenden.

  Auch hier soll wiederum eine `ToString` Methode implementiert werden, sodass wiederum eine hübsche Ausgabe möglich ist.

Schaue dir bei dieser Gelegenheit gleich das Konzept der Properties in C# genauer an und übe diese (d.h. programmiere auch selber ein Property aus, also nicht auto-implemented).

- Die Funktionaliät des Lexers wird durch das folgende Interface `ILexer` beschrieben:

```
namespace <matnr>.lexer {
  public interface ILexer {
      void add_definition(Definition def);
      IEnumerable<Token> tokenize(string source);
  }
}
```

- Jetzt zur Implementierung der Klasse `Lexer`, die die konkrete Implementierung des Lexers enthalten soll.

  Beachte dazu die allgemeinen Ausführungen zur Implementierung eines Lexers im Abschnitt Lexer und die beiden nachfolgenden Tipps. Das Testen wollen wir im nächsten Punkt erledigen.

  – Abgesehen von den Erläuterungen von vorher, hier noch ein spezieller Tipp zur Implementierung der Funktion `IEnumerable<Token> tokenize(string source)`. Der Sinn dieser Signatur ist natürlich, dass sich der Parser die einzelnen Tokens in weiterer Folge genauso einfach abholen kann wie dies beim Durchlaufen einer Schleife über ein Array oder eine Liste möglich ist. Warum aber dann nicht einfach ein Array von Tokens oder eine Liste von Tokens anlegen und einfach über die Sequenz iterieren? Weil zuerst die gesamte Datenstruktur im Speicher angelegt werden müsste.

    Die Lösung liegt daran, dass man ein `IEnumerable<Token>` zurückliefert und intern *kein* Array und auch *keine* Liste anlegt. Aber wie soll das funktionieren? Hier kommt die Anweisung `yield` von C# ins Spiel (die es in anderen Programmiersprachen in ähnlicher Form genauso gibt):

    ```
    yield return new Token(...);
    ```

    Damit wird erstmalig ein `IEnumerable` erzeugt und zurückgeliefert. Dieses kann dann z.B. in einer `foreach` Schleife verwendet werden, da der Compiler spontan einen Aufruf `getEnumerator()` generiert und mittels dieses Objektes und den Methoden `MoveNext()` bzw. dem Property `Current` auf das aktuelle Objekt (bei uns `Token`) zugreifen kann.

    Beim nächsten Aufruf von `MoveNext()` wird wieder in die Funktion `MoveNext` an die letzte Stelle (an der mittels `yield` zuletzt die Funktion verlassen wurde) gesprungen und dort weiter gemacht!

    Prinzipiell kann man einen `IEnumerator` natürlich auch ohne `yield` programmieren (als "Studierbeispiel" habe ich `./IdStore100.cs` vorbereitet).

– Weiters noch ein paar Hinweise zum Umgang mit regulären Ausdrücken in C#:

  * Es eignen sich besonders verbatim string Literale in der Form von `@"ein Backslash \ und ein "" (doppeltes Anführungszeichen)".`= Beachte die Verwendung von `\` und das doppelte `"` (da `\` keine besondere Bedeutung hat).

  * `System.Text.RegularExpressions` enthält die Klasse `Regex`, die im Konstruktor den eigentlichen regulären Ausdruck als String erwartet. Es *kann* noch ein zweites Argument übergeben werden, bei dem es sich um eine Enumeration vom Typ `RegexOptions` (die Definition enthält das Attribut `Flags`) handelt. Hier ein Beispiel:

    ```
    // eine geöffnete runde Klammer
    Regex re=new Regex(@"\(", RegexOptions.Compiled);
    ```

    Die Methode `Match(source, start_idx)` liefert ein "match"-Objekt zurück, das die folgenden Properties aufweist:

    · `Success` → `true`, wenn ein Muster gefunden, anderenfalls `false`

    · `Index` → Index an dem das gefundene Muster beginnt

    · `Length` → Länge des gefundenen Musters

    Und damit es etwas leichter wird folgt hier noch der String, der den regulären Ausdruck eines Symbols (ohne die vordefinierten Symbole wie `<`, `<=`,…) angibt: `@"([\w-[0-9]]\w*)|[+/*-]"`. Versuche diesen regulären Ausdruck zu entziffern!

    Eine Spezialität (d.h. in anderen Implementierungen nicht üblich) ist in diesem regulären Ausdruck enthalten, das direkt auf ein nützliches Feature der .Net Implementierung von regulären Ausdrücken zurückgreift: Es kann in einer Zeichenklasse eine "Subtraktion" (also mengenartige Differenz) spezifiziert werden:

    ```
    [a-z-[aeiou]]
    ```

    Jedes ASCII-Zeichen von `a` bis `z`, jedoch *ohne* die Zeichen, die in der Zeichenklasse `[aeiou]` enthalten sind!

    Den Rest musst du selbst analysieren können! Als Hilfestellung (auch zur Implementierung der restlichen regulären Ausdrücke) habe ich das offizielle Referenzsheet von Microsoft beigelegt.

* Das Zeilenende wird entweder als `\r\n` (Windows, Internet), `\n` (Unix) oder `\r` (Uralt Mac) markiert (wichtig, um die Zeilen- und Spaltennumer zu finden). Diese Information benötigst du, um ein Zeilenende zu erkennen, also nicht bei der Definition der Tokens, sondern bei der Implementierung des Lexers.

So, jetzt hast du alle notwendigen Informationen, die du zur Realisierung deines Lexers benötigst. Happy hacking!!!

b) Teste deinen Lexer gleich an Hand unserer elispy-Sprache, d.h. lege die entsprechenden Tokendefinitionen sowie eine Instanz deines Lexers in deinem Hauptprogramm an und lasse dir die Tokens für `(+ 1 2 "abc def")` anzeigen (`LPAREN`, `SYMBOL`, `INTEGER`, `INTEGER`, `STRING`, `RPAREN`)!

Wenn dies funktioniert, dann teste weiters mit `'(+ 1 2)`. Hier sind die Tokens: `QUOTE`, `LPAREN`, `SYMBOL`,…

Wenn dies funktioniert, dann teste weiters mit `(name _n_ame _ < <= == > >=)` (`LPAREN`, `SYMBOL`, `SYMBOL`, `SYMBOL`,…).

Wenn dies funktioniert, dann teste weiters mit `(- 1 \n 2 \r\n 3)`. Hier liegt der Schwerpunkt auf `\n` (Unix) und `\r\n` (Windows) sowie der korrekten Berechnung der Positionen

Wenn dies funktioniert, dann teste mit `(+ 1 \n 3)` (2 Leerzeichen vor 3) und auch mit `(+ 1 \r\n 3)`. Hier geht es "nur" um die korrekte Berechnung der Positionen.

Wenn dies funktioniert, *dann* weiter.

c) Teste deinen Lexer jetzt mit `1.5`! Das sollte einen Fehler produzieren. Lege dafür eine Klasse `LexerException` an, die von `Exception` ableitet und sich ebenfalls im Namensraum `<matnr>.lexer` befindet. Teste jetzt deinen Lexer erneut!

d) Jetzt ist es natürlich für unsere spezielle Sprache elispy nicht sinnvoll, eine Instanz von `Lexer` anzulegen und danach die entsprechenden `add_definition` manuell Aufrufe vorzunehmen, wie dies bisher im Hauptprogramm stattgefunden hat.

Führe ein Refactoring durch, sodass es eine Klasse `SexpsLexer` im Namensraum `<matnr>.elispy` (**!**) gibt, die von `Lexer` *abgeleitet* ist und deren einziger Zweck es ist, im Konstruktor die richtigen `add_definition` Methodenaufrufe zu tätigen. Damit bleibt die Funktionalität wieder gleich, aber es gibt einen neuen Typ, der konkret einen Lexer für unsere Programmiersprache implementiert.

Diese Klasse kann in weiterer Folge in unserem Parser verwendet werden.

e) Führe ein weiteres Refactoring durch: Bis jetzt haben wir alle Tokens durch einen String identifiziert, nämlich `"STRING"`, `"SYMBOL"`,… Bei den Strings kann man allerdings leicht vertippen, sodass eine Überprüfung durch den Compiler mittels Typen eine gute Sache wäre.

   Wir *könnten* statt dessen auch ein `enum` verwenden. Allerdings verlieren wird dadurch an Flexibilität (z.B. neue Tokens dynamisch hinzuzufügen).

   Daher führen wir statt dessen für unseren `SexpsLexer` im Namensraum `elispy` eine `static` Klasse `Tokens` hinzu, die alle Tokennamen als Stringkonstanten enthält. In weiterer Folge werden wir nur mehr diese Stringkonstanten für unseren elispy-Lexer und auch den folgenden Parser verwenden.

```
public static class Tokens {
    public const string LPAREN=...
    ...
}
```

f) Ein letztes Refactoring noch und dann haben wir den Lexer fertig: Füge eine Methode `test` zum `SexpsLexer` hinzu, der den Testcode enthält. Der Aufruf der Methode `test` soll in weiterer Folge im Hauptprogramm auskommentiert werden, da wir diesen jetzt nicht mehr weiter benötigen.

Lexer fertig, alles gut!

4.2 Parser

Der nächste Schritt, der Parser!

Alles was mit dem eigentlichen Parser zu tun hat, kommt in den Namensraum `<matnr>.elispy` (wie schon die Klasse `SexpsLexer`), da es speziell auf unsere Programmiersprache elispy zugeschnitten ist. Einen recursive-descent Parser kann man halt nicht für jede beliebige Programmiersprache verwenden, nur für die Programmiersprache für die der Parser entwickelt worden ist. Damit unterscheidet sich dieser von unserem implementierten Lexer, der allgemein verwendbar ist.

Hier sind jetzt konkretere Angaben wie so ein Parser implementiert werden kann/soll/muss:

1. Implementiere jetzt den Parser… und beginne wieder mit einem Interface!

```
using System;
using System.Collections.Generic;

namespace <matnr>.elispy {
  public interface IParser {
      void parse(string source);
  }
}
```

Ok, das war einfach.

2. Schreibe dazu eine Klasse `SexpsParser` im Namensraum `<matnr>.elispy`, die das Interface `IParser` implementiert und in *weiterer* Folge mittels einer entsprechenden Instanz von `SexpsLexer` in der Lage sein wird, unsere Sprache zu parsen und für jede Produktionsregel eine geeignete Ausgabe auf der Konsole tätigen wird.

   Das bedeutet, dass du vorerst lediglich die Klasse mit den etwaigen Instanzvariablen und den Konstruktor implementieren sollst. Alles Weitere wird in den folgenden Punkten beschrieben und auch implementiert.

   Ok, tut nicht viel, aber es bringt uns einen (kleinen) Schritt weiter.

3. Jetzt kommen wir langsam dazu, unseren Parser konkret zu implementieren. Was soll der Parser eigentlich parsen? Wie sieht die Sprache aus? Wie sieht die Grammatik der Sprache aus?

   Erstelle daher die Grammatik in EBNF für eine sexp, wobei wir hier nur die abgekürzte Form einer sexp unterstützen wollen, d.h. folgende Beispiele für sexps für nicht-Atom-sexps (also Listen) sind:

   - `()`, d.h. eine leere sexp (d.h. gleich zu `nil`)
   - `(x)`, d.h. eine sexp, die eine sexps enthält
   - `(x y)`, d.h. eine sexp, die zwei sexps enthält
   - …

   Die eigentliche Form von sexps wird also *nicht* unterstützt (unser Lexer kennt ja auch nicht einmal den Punkt…). Für uns sind das einfach Listen.

   Zur Erinnerung: Eine sexp kann entweder ein Atom, eine Liste oder eine sexp sein, der ein Quote-Zeichen (`'`) vorangestellt ist. An Atomen kennen wir einen String, eine Zahl und ein Symbol. Mehr nicht!

Also nimm einen Zettel, mache dich, wenn notwendig, noch einmal mit der Syntax und Semantik von EBNF in der *ISO Variante* vertraut (siehe Folien zu Compilertechnologie) und entwerfe eine Grammatik für eine sexp. Die Tokens brauchen nicht weiter spezifiziert werden, hier reicht es diese zur Gänze in Großbuchstaben zu schreiben, um diese als Token zu kennzeichnen (unsere Konvention), also `INTEGER`, `STRING` und `SYMBOL`.

Erweitere die Grammatik, sodass diese ein elispy-Programm, also eine *Liste* von sexps, beschreibt. Achtung: Damit meinen wir jetzt nicht das Nonterminalsymbol, das eine Liste im Sinne einer sexp darstellt, sondern eine beliebige Anzahl im Sinne der EBNF! Dabei handelt es sich um eine Entwurfsentscheidung!!

Hinweis: Die gesamte Grammatik kommt mit 4 Nonterminalsymbolen und 3 Tokens aus (also abgesehen von `(`, `)` und `'`). Die Tokens werden ja *nicht* in der Grammatik beschrieben, denn dafür haben wir ja unseren Lexer.

Schreibe diese Grammatik jetzt als Kommentar an den Anfang der Klasse `SexpsParser`!

Ok, wieder etwas geschafft (auch wenn keine Zeile Code programmiert worden ist).

4. Bevor wir die eigentliche Implementierung jetzt starten, implementiere eine Utility-Klasse `Utility` mit folgendem Inhalt in einer eigenen Datei, sinnvollerweise im Namensraum `<matnr>`:

```
public static bool In<T>(this T item, params T[] list) {
    return list.Contains(item);
}
```

Wichtig ist, dass du ein `using System.Linq;` verwendet hast. Erklärung folgt nachfolgend.

Was ist das? Hier wird eine ”Extension Method” definiert, die selber wiederum auf LINQ (Language INtegrated Query, eine Funktionalität von .Net) zugreift. Damit ist jetzt folgendes möglich:

```
String x="a";
if (x.In("a", "b", "c")) {}
```

D.h. es wird eine Methode definiert, die einen Typ erweitert (→ Generics), sodass dieser mit einer Method `In` verwendet werden kann. `this` gibt den Typ an, der erweitert wird und `params` gibt an, dass die übergebenen Argumente (variable Anzahl an Argumenten) als formaler Parameter `list` zur Verfügung stehen. Im Rumpf der Methode wird auf die Erweiterungsmethode (!) `Contains` aus `System.Linq` zugegriffen.

D.h. mit Erweiterungsmethoden kann einer schon bestehenden Klasse eine Methode hinzugefügt (also erweitert) werden, ohne die Klasse selber verändern zu müssen. Dies ist ein sehr sinnvolles Werkzeug, da man schon bestehende Klassen (z.B. aus der .Net oder aus einer third-party Bibliothek) gar nicht verändern kann.

5. Implementiere *jetzt* einen recursive-descent Parser für deine Grammatik in einer Klasse `SexpsParser.`

   Der Konstruktor soll folgendermaßen aussehen:

   ```
   public SexpsParser(ILexer lexer) {
       ...
   }
   ```

   Wir sehen, dass wir eine Lexerinstanz übergeben. Das wäre an sich nicht unbedingt notwendig, denn der Parser könnte sich selber eine Instanz anlegen. Allerdings ist es dies sehr vernünftig, denn dann können wir dem Parser eine beliebige Instanz unterschieben, solange dieser von `ILexer` abgeleitet ist. Damit können wir auch eine bestehende Instanz eines Lexers einfach weiter verwenden.

   Vorerst soll der Parser lediglich alle geparsten Tokens auf der Konsole *ausgeben*. D.h. direkt im Parser stehen an den entsprechenden Stellen entsprechende `Console.Write`-Aufrufe. Das ist natürlich nicht besonders gut, aber für den Anfang reicht es.

   Damit werden die Funktionen, die die Produktionsregeln unserer Sprache in Form eines recursive-descent Parser realisieren (vorerst) keinen Rückgabewert zurückliefern.

   Weiters soll der Parser alle Fehler *zuverlässig* melden. Fehler sollen mittels einer Exception dem Aufrufer der Methode `parse` zur Verfügung gestellt werden. Schreibe dazu eine Klasse `ParserException` analog zur Klasse `LexerException` im Namensraum `<matnr>.elispy`.

   Der Aufruf der Methode `test` des Lexer wird in weiterer Folge nicht mehr benötigt und soll daher auskommentiert werden. Damit kann dieser bei Bedarf (d.h. zum Testen) wieder aktiviert werden und steht außerdem zu Dokumentationszwecken im Sourcecode.

   Wenn dies funktioniert, dann ist ein wichtiger Schritt getan (auch wenn eigentlich noch nicht so viel passiert ist).

6. Nachdem der Parser prinzipiell korrekt parst, aber sonst, wie schon festgestellt, noch nichts Vernünftiges tut, ist es an der Zeit dies zu ändern.

So, was ist eigentlich noch die Aufgabe eines Parsers, außer, dass dieser die Grammatik überprüft und Fehler meldet? Er soll einen AST (abstract syntax tree) erstellen und eine Symboltabelle befüllen (siehe Folien zu Compilertechnologie). Die Symboltabelle wird in unserem konkreten Fall nicht befüllt, aber beim Auswerten eines sexp-Ausdruckes werden wir diese später benötigen.

Ein AST stellt das geparste Programm mit allen wesentlichen Elementen dar. Im Gegensatz dazu gibt ein Syntaxbaum (oder concrete syntax tree) die konkrete Syntax eines Programmes wieder. In unserem Fall gibt es eigentlichen keinen großen Unterschied.

Wir gehen dies so an, dass die Methode `parse` einen AST zurückliefern wird und deshalb wird die Signatur der `parse`-Methode des Interfaces `IParser` folgendermaßen abgeändert:

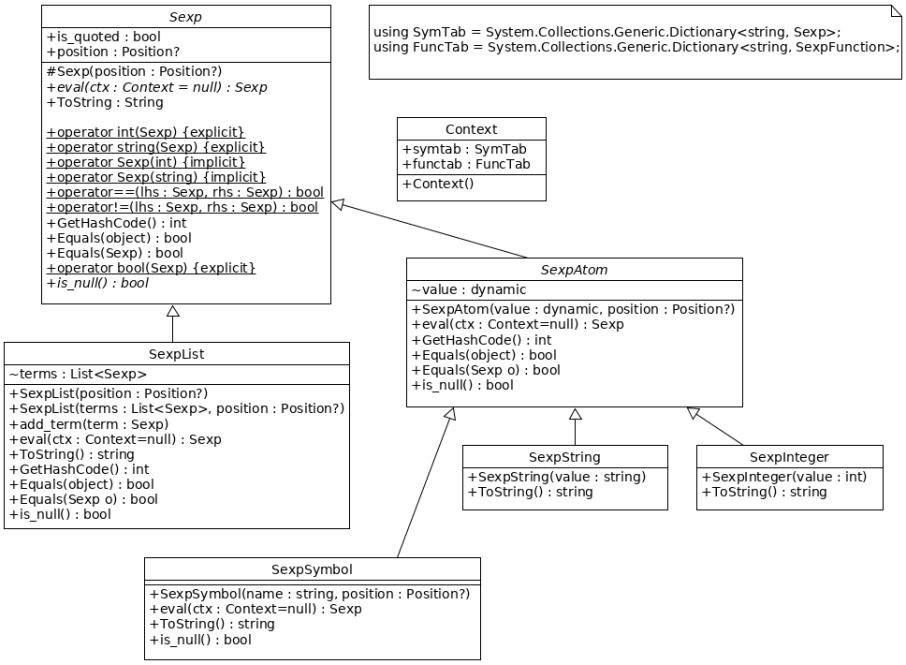
```
List<Sexp> parse(string source);
```

Hmm, ok, wenn man sich das ansieht, dann ist das nicht *ein* AST sondern eine Liste von `Sexp` (soll eine sexp repräsentieren). Jede Instanz von `Sexp` stellt für sich einen AST für die repräsentierte sexp dar.

Wenn man wollte, könnte man ein elispy-Programm als eine einzige Listen-sexp darstellen, wir sehen das aber nicht so. Für uns ist ein elispy-Programm eine Folge von sexps und in diesem Sinne liefert die Funktion `parse` eben auch eine Liste von `Sexp` zurück.

Die eigentliche Implementierung von `parse` verschieben wir wieder etwas auf später (Dummy-Implementierung nicht vergessen → Compiler will zufrieden sein).

`List<Sexp>` ist also eine Liste von Instanzen von `Sexp`. Wie sieht allerdings ein `Sexp` aus? Welche Arten gibt es? Wie sehen diese aus? Welche Methoden sind sinnvoll? Gar nicht so einfach! Daher folgt hier ein Klassendiagramm, das diese Fragen beantwortet und von dir so zu implementieren ist:



Das Klassendiagramm verwendet der einfacheren Umsetzung in ein Programm direkt C#-Syntax (abgesehen von der UML-spezifischen Syntax). Folgende Attribute und Methoden sind enthalten:

- Das Attribut `is_quoted` kennzeichnet, dass die aktuelle sexp mit dem Quote-Operator versehen wurde. D.h., dass wir die Verwendung des Quote-Operators nicht direkt im AST wiederfinden wollen. Auch das ist eine Designentscheidung!

- Das Attribut `position` gibt die Position dieser sexp an. Dies wird benötigt, um in weiterer Folge Fehlermeldungen bereitstellen zu können. Eh klar.

- Die Methode `eval` soll in *weiterer* Folge eine sexp auswerten. Wir werden das vorerst einmal auf später verschieben. Fülle die Rümpfe lediglich so, dass der Compiler zufrieden ist.

  Wir sehen, dass hier u.U. eine Instanz der Klasse `Context` übergeben wird. Diese Klasse beinhaltet sowohl eine Symboltabelle (Dictionary, das den Namen des Symbols und den referenzierten sexp enthält) und auch eine Funktionstabelle. Die Funktionstabelle wird erst bei der Implementierung des Interpreters verwendet und wird vorerst aus der Klasse `Context` weggelassen (d.h. im Moment nicht implementieren!).

  Weiters sehen wir eine UML-Notiz, die eine *using alias directive* von C# verwendet und einen Alias für einen (langen) Typnamen festlegt. Das ist zwar

an sich nicht notwendig, aber man will so etwas ja auch einmal programmiert haben.

Damit dieser Context zur Verfügung steht, werden wir den Konstruktor des Parsers leicht modifizieren:

```
public SexpsParser(ILexer lexer, Context ctx) {
    ...
}
```

Dazu muss sich der Parser natürlich seinen Context abspeichern und beim Aufruf eine entsprechende Instanz mitgegeben werden.

- Die Methode `ToString` soll natürlich eine entsprechende elispy-Darstellung des Sexps zurückliefern. Das ist wichtig, dass dies korrekt implementiert wird, da es uns helfen wird, die Korrektheit unseres Parsers zu beurteilen.

  Bei `SexpSymbol`, `SexpInteger` und `SexpString` ist dies relativ leicht und soll in weiterer Folge (also hier vorerst nur die Beschreibung, die eigentliche Implementierung folgt in Kürze) so implementiert werden:

  - `new SexpSymbol("a")".ToString()` → `a`
  - `new SexpInteger(1).ToString()` → `1`
  - `new SexpString("abc").ToString()` → `"abc"`

  Wenn die Sexp allerdings ”quoted” ist, dann sollte es zu solchen Ergebnissen kommen:

  - `s = new SexpSymbol("a"); s.is_quoted = true; s.ToString()` → `'a`
  - analog für `'1` → `'1`
  - analog für `'"abc"` → `'"abc"`

  Das schreit förmlich danach, dass das Hinzufügen des Zeichens `'` in der Klasse `Sexp` erledigt wird!

  Listen funktionieren in genau der gleichen Art und Weise. Eine kleine Besonderheit ist, dass wir eine leere Liste gleich als `nil` zurückgeben werden.

  - `new SexpList(new List<Sexp>()).ToString()` → `nil`
  - eine Liste mit den Werten 1, 2, und 3 → `(1 2 3)`

Auch hier sind natürlich "gequotete" Listenausdrücke korrekt anzuzeigen:

- `'()` → `'()`
- `'(1 2 3)` → `'(1 2 3)`

So, das wäre in Ordnung, aber in Lisp wird eine leere Liste als `nil` dargestellt! Daher ist die Methode `ToString()` der Klasse `SexpList` entsprechend zu implementieren:

- `'()` → `'nil`

Ein gequoteter Ausdruck `nil` wird natürlich in unserem Alltag nicht oft auftreten, da `'()` zu `()`, also `nil`, evaluieren wird…

- In der Klasse `SexpList` findest du einen überladenen Konstruktor. Nichts besonderes, aber… Einmal wird eine leere Liste angelegt und einmal wird die übergebene Liste verwendet.

  An sich nichts besonderes, aber achte, darauf dass die Liste unter keinen Umständen zwei Mal angelegt wird. Das würde sich ungünstig auf die Speicherverwaltung und auch die Laufzeit auswirken. Ok, in diesem konkreten Fall wären die Auswirkungen nicht dramatisch, aber es geht ja um das Prinzip!

- Im Klassendiagramm noch weitere Operatoren und die Methoden `Equals`, `GetHashCode` und `is_null` angeführt, die wir jetzt einmal in gewohnter Weise ignorieren (d.h. nicht implementieren)! Auch später wollen wir noch Spaß haben.

Implementiere jetzt die Klasse `Context` (ok, enthält eigentlich nicht viel) und danach wage dich an die Klassenhierarchie der sexp in der Datei `Sexps.cs`. Weil diese Klassen doch zusammengehören und eine Änderung an einer Basisklasse oft Änderungen an den Kindklassen nach sich ziehen und es außerdem übersichtlicher ist, werden wir eine Ausnahme von unserer Regel vornehmen (Ausnahmen bestätigen ja bekannterweise die Regel) und alle Klassen in *einer* Datei `Sexp.cs` speichern.

Teste zumindest die `ToString` Methode durch entsprechende Aufrufe und Ausgaben in der Methode `test()` der Klasse `SexpsParser`.

Hmm, wieder kein funktionsfähiger Parser entstanden… Macht nichts, weiter mit dem nächsten Punkt.

7. Jetzt müssen die Funktionen, die die einzelnen Produktionsregeln implementieren, angepasst werden, damit ein AST erzeugt werden kann. Baue daher deine Funktionen der Produktionsregeln auf folgende Art um (natürlich kann es sein, dass deine Produktionsregeln anders heißen, aber… Namen sind ja bekanntlich Schall und Rauch):

```
// wie auch immer du deine Hauptregel genannt hast, die ein
// elispy-Programm repräsentiert...
private List<Sexp> program() {
    ...
}

private Sexp sexp() {
    ...
}

private SexpAtom atom() {
    ...
}

private SexpList list() {
    ...
}
```

Jetzt ist es an der Zeit, dass diese so angepasst werden, dass diese zum Funktionieren gebracht werden. Jetzt!

8. Nun ist es an der Zeit zu testen: Teste jetzt wieder deinen Parser indem du dir den Rückgabewert von `parse` (also der Rückgabewert von `program`) auf der Konsole ausgeben (`ToString` einer `Sexp` ist implementiert!) lässt (also bei einer `List<Sexp>` derzeit natürlich nur das Element mit dem Index 0), wenn wir nur eine einzige sexp parsen.

   Teste zumindest mit den folgenden Teststrings, wobei die Ausgabe genauso wie die Eingabe aussehen sollte (abgesehen von Whitespace Zeichen):

   - `(+ 1 2 name "abc def")`
   - `'(+ 1 2)`
   - `( + 1 2 )` … Ausgabe hier natürlich *ohne* die überflüssigen Leerzeichen

   Für die folgenden fehlerhaften Eingaben sollten diese und die entsprechenden Fehlermeldungen (in etwa so) ausgegeben werden:

   - `(+ 1 2 Missing ')' or EOF at (index=5, line=1, column=5)`
   - `(.+ 1 2 ) Unrecognized symbol '.' at (index=1, line=1, column=1)`
   - `( Opening '(' but EOF at (index=0, line=1, column=0)`

Jetzt haben wir aber wirklich schon viel erreicht, denn unser Parser parst richtig und erstellt auch einen korrekten AST (hoffentlich, da nicht direkt, sondern nur indirekt über die Ausgabe, getestet). Auch die textuelle Repräsentation haben wir gut hingekriegt. Ein bisschen Eigenlob *hie* und *da* schadet nicht.

Die Testausgaben, die wir direkt im Parser in den Funktionen der Produktionsregeln geschrieben haben, können jetzt wieder entfernt werden, da wir diese jetzt nicht mehr benötigen! Wir haben ja jetzt eh die Rückgabewerte.

Und damit unser Hauptprogramm nicht so mit Testcode zugepflastert ist, werden wir in der Klasse `SexpsParser` wieder eine Methode `test()` schreiben und diese dorthin verschieben. Jetzt noch ein Aufruf der Methode `test()` im Hauptprogramm. Es soll genauso wie zuvor funktionieren – ein klassischer Fall von Refactoring!

Was jetzt noch fehlt ist die Auswertung samt der gesamten Logik, aber das ist den nächsten Schritten vorbehalten.

Vorerst können wir voller Stolz verkünden, dass unser Parser funktioniert und einen AST erzeugt (beides hoffentlich korrekt). Um diesen wirklich zu testen, müssten wir den AST auch richtig testen (z.B. mittels Unittests). Aber das ist eine andere Geschichte!

4.3 Interpreter

Jetzt weiter zum eigentlichen Hauptteil unseres Miniprojektes, nämlich dem Interpreter!

1. Beginnen wir wieder mit einer eigenen Klasse `SexpsInterpreter`, die über einen einfachen Konstruktor mit folgender Signatur verfügen soll:

   ```
   public SexpsInterpreter(SexpsParser parser)
   ```

   Und weil wir u.a. wieder Testcode in einer eigenen Methode `test` ablaufen lassen wollen, werden wir solch eine Methode jetzt einmal leer implementieren, in unser Hauptprogramm einbauen und später mit Leben befüllen.

   Aber im Moment gibt es ja noch nichts zu testen, also weiter mit dem nächsten Punkt.

2. Dazu muss in einem nächsten Schritt unsere `Sexp`-Hierarchie noch weiter mit Leben befüllt werden.

a) Beginnen wir in einem ersten Schritt mit der Funktion `eval`. Wie eine sexp auszuwerten ist, ist prinzipiell den `Lisp`-Folien zu entnehmen! Nur die Implementierung von `eval` einer `SexpList` ist etwas komplizierter und in gewohnter Weise werden wir diese auf später verschieben (an sich kein optimaler Ansatz die schwierigen Aufgaben zu verschieben, aber…).

D.h., wenn wir `SexpList` einmal *nicht* betrachten, dann sind vorerst nur die beiden folgenden Fälle zu betrachten:

- Ein Atom evaluiert prinzipiell zu sich selbst, *außer* es handelt sich um ein Symbol (siehe später). Das können wir jetzt sofort programmieren, das ist ja wirklich kein Problem.

  Das sollte jetzt schon funktionieren und kann auch mit den folgenden Testdaten (in der Methode `test()`) getestet werden:

  ```
  Console.WriteLine(parse("1")[0].eval());
  Console.WriteLine(parse("\"abc\"")[0].eval());
  Console.WriteLine(parse("'1")[0].eval());  // -> quoted!
  Console.WriteLine(parse("'\"abc\"")[0].eval());
  Console.WriteLine(parser.parse("'a")[0].eval());  -> quoted!
  ```

  Die Ausgabe muss folgendermaßen aussehen:

  ```
  1
  "abc"
  1
  "abc"
  a
  ```

  Beachte im Speziellen die letzte Ausgabe! Während es bei einer ganzen Zahl oder einem String unerheblich ist, ob dieser ”quoted” ist oder nicht, ist dies bei einem Symbol nicht so…

- Ein Symbol evaluiert so, dass in der Symboltabelle nachgeschlagen wird und der dort gespeicherte Ausdruck zurückgeliefert wird. Ist das Symbol nicht in der Symboltabelle gespeichert, dann soll zuverlässig ein entsprechender Fehler gemeldet werden. Und weil wir uns jetzt schon im Interpreter-Teil befinden, wollen wir gleich auch eine neue Klasse `InterpreterException` implementieren.

  Mit folgendem Testcode

  ```
  try {
      Console.WriteLine(parse("a")[0].eval());
  ```

```
} catch (Exception e) {
    Console.WriteLine(...);
}
```

sollte es in etwa zu folgender Ausgabe kommen, da wir noch über kein Symbol `a` verfügen:

```
a               Symbol "a" not defined at (index=0, line=1, column=0)
```

Das ist ja auch klar, da wir ja noch gar keine Symboltabelle angelegt haben. Lege dazu eine Instanz von `Context` in der Methode `test` an und gib diese in weiterer Folge jeweils an `eval`.

Füge nun zu Testzwecken in die angelegte Symboltabelle *manuell* einen Eintrag für das Symbol `a` hinzu.

Kontrolliere dies jetzt indem du dir alle Symbole (also im Moment genau eines) samt den dazugehörigen Werten ausgeben lässt. So vielleicht?

```
foreach (var x in ctx.symtab)
    Console.WriteLine(x);
```

Parse jetzt den gleichen Ausdruck nochmals, aber diesmal richtig:

```
foreach (var x in ctx.symtab)
      Console.WriteLine(x);
Console.WriteLine(parser.parse("a")[0].eval(ctx));
```

Das sollte jetzt funktionieren!!!

Bis jetzt sollte das Programm wieder ohne Fehler übersetzen und auch soweit funktionieren.

b) Bevor wir uns an die Implementierung von `eval` der Klasse `SexpList` heranwagen, werden wir uns mit den Symbolen beschäftigen. Es gibt die vordefinierten Symbole und die Symbole, die im Zuge der Abarbeitung eines elispy-Programmes hinzugefügt werden.

Die einfachste Variante ist, die vordefinierten Symbole direkt im Programm zu implementieren und könnten daher auch nicht redefiniert werden. Und weil es der einfachste Weg ist, werden wir diesen *nicht* beschreiten. Trotzdem werden wir es uns nicht allzu schwer machen und deshalb werden wir keine *benutzerdefinierten* Funktionen zulassen.

Damit haben wir eine Entscheidung getroffen, dass vorerst an der allgemeinen Struktur nichts zu tun ist und können direkt zum nächsten Punkt gehen `:-)`

c) Jetzt zur Implementierung von `eval` in der Klasse `SexpList`. Ok, die ist etwas komplizierter. Teilen wir dies wieder etwas auf, dann wird es einfacher:

- Wenn es sich um einen Listenausdruck handelt, der "quoted" ist, dann wird der eigentliche Listenausdruck unverändert zurückgeliefert. Füge in der Methode `test` folgende Anweisung hinzu:

  ```
  Console.WriteLine(parser.parse("'(1 2 3)")[0].eval());
  ```

  Hier muss `(1 2 3)` ausgegeben werden!

- Wenn die Liste leer ist, dann muss gemäß der Spezifikation von Lisp der Ausdruck zu `nil` evaluieren. Hmm, dazu müssten wir einmal wissen woher wir `nil` nehmen sollen. Tja, da es sich um ein Symbol handelt und wir keine Hardcore-Programmierung vornehmen wollen (errinnere dich, das war eine Designentscheidung), müssen wir die Symboltabelle initialisieren. Gemäß der Programmiersprache Lisp, handelt es sich bei `nil` um ein Symbal, das der leeren Liste entspricht. Hier gibt es an sich die Möglichkeit, dies als eine Intanz von `SexpSymbol` oder als eine Instanz von `SexpList` zu implementieren. Wir entscheiden uns für die zweite Möglichkeit, da es sich als einfacher herausstellen wird.

  Wo? Natürlich im Konstruktor der Klasse `Context`. Also, so etwas wie die folgende Anweisung würde sich eben dort gut machen:

  ```
  symtab["nil"] = new SexpList(new List<Sexp>());
  ```

  Ok? Teste wieder entsprechend in `test`!

  Und weil es so nett ist, füge gleich ein weiteres Symbol, nämlich `t` zu der Symboltabelle hinzu. Hier stellt sich die Frage, wie dies zu implementieren ist… Hier gehen wir einen anderen Weg und werden dies folgendermaßen umsetzen:

  ```
  symtab["t"] = new SexpSymbol("t");
  ```

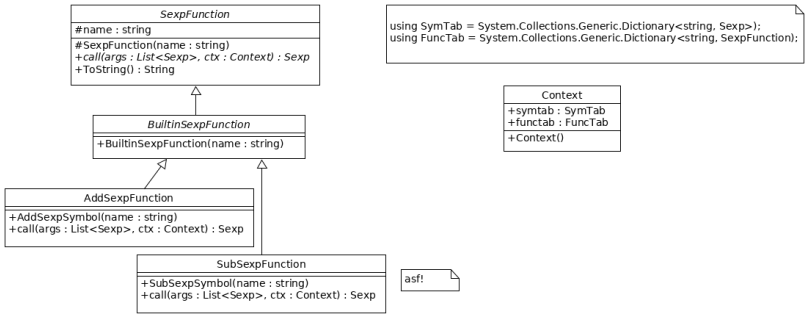
- Ist die Liste nicht leer, dann muss sich an erster Stelle ein `SexpSymbol` befinden, anderenfalls ist dies wiederum ein Fehler.

  ```
  try {
      parser.parse("(1 2 3)")[0].eval(ctx);
  } catch (InterpreterException e) {
      Console.WriteLine(...);
  }
  ```

  Ein Ausgabe etwa in folgender Form wäre wieder nicht so verkehrt:

```
(1 2 3)        First item must be a symbol, but got "1" at (index=0, line=1,
```

- So, jetzt wollen wir an die Evaluierung von Ausdrücken der folgenden Gestalt `(+ 1 2)` bzw. `(+ 1 (+ 2 3))` wagen. Dazu benötigen wir natürlich eine Implementierung dieser Funktionen. Hmm, also zuerst benötigen wir ein Modell in Form eines UML-Klassendiagramms… Here you are!



  Implementiere jetzt die Klassen `SexpFunction` und `BuiltinSexpFunction` in einer Datei `SexpFunctions.cs`. Auch hier werden wir, analog zu den verschiedenen Sexps, die gesamte Klassenhierarchie in einer Datei implementieren!

  Nebenbei wäre noch Platz für eine Klasse `UserDefinedSexpFunction`, aber wir haben uns entschieden, diese nicht zu implementieren. Aber wenn dir unbedingt danach ist, kannst du es dir ja für später aufheben. Es liegt nicht in meiner Natur deinen Tatendrang einzubremsen. Wer die Herausforderung annimmt, soll mir das funktionierende Endergebnis zeigen! Ich werde es wohlwollend bewerten `;-)`

- Weiter geht es mit der ersten zu implementierenden Funktion, nämlich der Funktion zum Addieren von ganzen Zahlen. Die Klasse nennen wir `AddSexpFunction`.

  Der Konstruktor ist eh klar und deshalb implementiere diese jetzt und sofort.

  Die Function `call` ist da schon etwas schwieriger. Hier nochmals eine kurze Funktionsbeschreibung:

```
(+) ; -> 0
(+ 1) ; -> 1
(+ 1 2) ; -> 3
(+ 1 2 3) ; -> 6
```

  Und jetzt noch eine Minianleitung:

– Schreibe zuerst in der Klasse `Sexp` einen Operator `int`:

```
public static explicit operator int(Sexp sexp) {
    // ...
}
```

Das übergebene Argument muss eine Instanz von `SexpInteger` sein, anderenfalls handelt es sich um einen Fehler. Verwende den `as` Operator. Liegt ein Fehler vor, dann werfe eine `ArgumentExpception`, da ja ganz offensichtlich das Argument nicht passend gewesen ist.

Liegt kein Fehler vor, dann soll `value` zurückgeliefert werden.

`explicit` bedeutet, dass der Operator von C# nicht implizit aufgerufen wird, sondern die Konvertierung explizit mittels `(int)` angefordert werden muss. Leider kann man `implicit` in diesem konkreten Fall nicht verwenden, da C# diesen sonst in einigen ungünstigen Fällen (eben implizit) aufruft (z.B. bei der Ausgabe mittels `WriteLine`) und dies zu einem Fehler führt.

Implementiere in gleicher Art und Weise gleich den Operator zur Umwandlung in einen `string`!

– Schreibe in der Klasse `Sexp` einen Operator `Sexp`, der einen `int` in eine Instanz von `Sexp` wandelt:

```
public static implicit operator Sexp(int value) {
    // ...
}
```

Die Implementierung ist trivial.

Implementiere in gleicher Art und Weise gleich den Operator zur Umwandlung eines `string` in eine Instanz von `Sexp`!

Jetzt sind wir in der Lage für unsere Belange beliebig zu konvertieren. Wir werden es noch benötigen.

– Jetzt zurück zur eigentlichen Implementierung der Funktion `call` von `AddSexpFunction`. Jeder Term der Argumente wird mittels `eval` *ausgewertet*, dann mittels dem Operater `(int)` in eine ganze Zahl konvertiert werden. In der `return` Anweisung kann die ganze Zahl direkt angegeben werden, da die Operatorfunktion zum Umwandeln in eine `Sexp` *implizit* vom Compiler aufgerufen wird!

Damit ist alles fertig was mit dem Addieren zu tun hat. Jetzt muss natürlich noch die entsprechende Behandlung einer `SexpList` samt dem korrekten Aufruf der `call` Methode kodiert werden. Darum kümmern wir uns gleich im nächsten Punkt.

- Ist die Liste nicht leer und befindet sich an erster Stelle ein `SexpSymbol`, dann muss in der `functab` ein entsprechendes Funktionsobjekt gesucht (und hoffentlich gefunden) werden, dann die `call` Methode aufgerufen werden (mit den restlichen Items der Liste und dem aktuellen Context als Argumenten). Das Ergebnis wird zurückgeliefert. Gar nicht so schwer, nicht wahr?

  Hmm, es könnte ja eine `ArgumentExpception` vom Konvertierungsoperator geworfen werden, aber wir hätten gerne einen `InterpreterException`, nicht wahr? Fange deshalb die `ArgumentException` in der Methode `eval` der Klasse `SexpList` ab und werfe eine `InterpreterException` mit der Instanz als `ArgumentException` als innere Exception! Damit kannst du auch gleich die richtige Position setzen (die ja in der Methode `call` nicht verfügbar ist).

- Ergänze bitte die Funktionstabelle um einen Eintrag für die Funktion `+` im Konstruktor der Klasse `Context`!

- Jetzt geht es an das Testen! Teste die folgenden Fälle in der Methode `test`:

  ```
  (+)
  (+ 1)
  (+ 1 2)
  (+ 1 2 3)
  (+ 1 (+ 2 3))
  (+ 1 "a")
  ```

3. Jetzt ist es an der Zeit die Kommandozeilenschnittstelle gemäß der Spezifikation (jedoch noch ohne konkrete Funktionalität der Option `-g`) zu implmentieren.

   Was notwendig ist, ist notwendig und deshalb werden wir im Zuge der Umsetzung der Kommandozeilenschnittstelle den etwaigen Testcode unter Kommentar zu setzen.

   Beginnen wir wieder mit dem einfachsten Teil und arbeiten wir uns sukzessive zu den schwierigem Teil zu:

   a) Wurde in der Kommandozeile als Dateinamen `-` mitgegeben, dann sollen die sexps von `stdin` gelesen werden. Das werden wir auf später verschieben.

b) Wurde in der Kommondozeile als Dateinamen ein konkreter Dateinamen angegeben, dann soll die Datei gelesen, geparst und alle sexps der Reihe nach ausgewertet werden. Auch das werden wir auf später verschieben.

c) Wurde in der Kommandozeile kein Dateinamen angegeben, dann soll das Programm als REPL-Interpreter (read-evaluate-print-loop) funktionieren.

   Dazu soll die Schnittstelle so funktionieren:

```
$ elispy
elispy> (+)
0
elispy> (+ 1)
1
elispy> (+ 1 2)
3
elispy> (+ 1 2 3)
6
elispy> (+ 1 (+ 2 3))
6
elispy>
```

   Zu sehen ist, dass nach dem Start des Programmes (ohne weitere Kommandozeilenargumente) ein Prompt ausgegeben wird und danach der Prozess auf eine Eingabe wartet (also so wie unter Python). Jede Eingabe wird ausgewertet und das Ergebnis ausgegeben. Das sollte nach Implementierung auch so funktionieren.

   Hier wieder wieder eine kleine Anleitung:

   - Schreibe eine Methode `repl` in der Klasse `SexpsInterpreter`, die genau diese Benutzerschnittstelle implementiert.
   - Lese die Benutzereingabe zeilenweise mittels `Console.ReadLine` ein.
   - Rufe damit die `parse` Methode von `SexpsParser` auf. Tritt ein Fehler auf, dann soll die Fehlermeldung auf der Konsole ausgegeben werden.
   - und beginne wieder von vorne.

d) *Wenn* das prinzipiell so funktioniert, dann ist es an der Zeit sich verschiedene Situationen anzusehen:

- Wie wird der Prozess beendet? Ok, man kann den Prozess im Taskmanager einfach beenden. Man kann einfach `CTRL-C` drücken. Aber das sind ziemlich krude Vorgehensweisen, nicht wahr?

  Drückst du aber `CTRL-D` (unter Linux, macOS), dann wird der Eingabestrom zum Prozess geschlossen. In diesem Fall liefert die `ReadLine`-Methode `null` zurück…

  Kümmere dich jetzt darum, dass dein Interpreter mittels `CTRL-D` beendert werden kann.

- Unnötige Zeichen sollten von Haus aus aus der Eingabe entfernt werden. Warum sollte man damit den Interpreter belasten?

  – Handelt es sich um unnötige Leerzeichen am Anfang und am Ende der Benutzereingabe, dann können diese einfach mittels der Methode `Trim` entfernt werden.

    Testen!

  – Schwieriger ist es wenn der Benutzer Steuerzeichen eintippt, wie z.B. `CTRL-P` (z.B. weil sich dieser vertippt). In diesem Fall würde unser Lexer einen Fehler liefern. Auch diese gehören entfernt.

    Das kann auf folgende Art und Weise erreicht werden:

    ```
    input = new string(input.Where(c => !char.IsControl(c)).ToArray());
    ```

    Hier greifen wir wieder auf Linq zurück! Analysiere diese Anweisung.

- Hat der Benutzer keine Eingabe getätigt (also der Leerstring wurde effektiv eingegeben), dann ist auch nichts zu tun und einfach den nächsten Schleifendurchgang zu beginnen.

e) Weiter mit der Kommandozeilenoption `-`. Das übliche Verhalten eines Programmes ist, dass es von `stdin` die Eingabe liest. Das funktioniert folgendermaßen:

```
$ echo "(+ 1 2) (+ 2 3)" | elispy -
5
```

D.h. das Programm startet, liest von `stdin` die Ausdrücke, evaluiert diese und schreibt (vorerst) das Ergebnis des letzten Ausdruckes nach `stdout`. Unser Programm erkennt dies, da als Kommandozeilenargument `-` übergeben worden ist.

In diesem Beispiel ist dies so, dass der Stream `stdout` des ersten Prozesses mit dem Stream `stdin` des zweiten Prozesses mittels einer Pipe verbunden worden ist.

Ein Tipp: `Console.In.ReadToEnd` ist eine Methode, die "alles" von `stdin` liest und als String zurückliefert. Das könnte doch helfen, dies korrekt zu implementieren.

Implementiere dieses Verhalten jetzt!

Beachte bitte, dass du das selbe Verhalten mit dem Kommando `dotnet run` nur auf folgende Art und Weise erhältst:

```
$ echo "(+ 1 2) (+ 2 3)" | dotnet run -- -
```

Du siehst, dass die eigentlichen Kommandozeilenargumente über `dotnet` nach zwei aufeinanderfolgenden Bindestrichen folgen müssen.

f) Und weil wir jetzt schon beim Implementieren der Benutzerschnittstelle sind, werden wir den letzten Teil, nämlich das Einlesen der Sexps aus einer Datei, implementieren.

Gut, das ist jetzt nicht besonders schwierig… Hier noch ein Beispiel:

```
$ elispy test.el
3
```

Wir gehen davon aus, dass in der Datei `test.el` nur die Sexps `(+ 1 1) (+ 1 2)` enthalten sind. Der Wert des letzten Ausdrucks wird (vorerst) ausgegeben.

Ein Tipp: `File.ReadAllText` ist wieder hilfreich. Aber nicht vergessen, dass auch hier Fehler auftreten können, wie eine nicht existente Datei oder fehlende Berechtigungen,… D.h. es ist nicht zu vergessen, dass eine etwaige Exception abzufangen ist und das Programm mit einer entsprechenden "usage"-Meldung zu beenden ist.

4. Implementiere in gleicher Art und Weise die folgenden Funktionen:

   - `-` … `SubSexpFunction`. Hier wieder eine kleine Demonstration, die das Verhalten spezifiziert:

     ```
     $ elispy
     elispy> (-)
     0
     elispy> (- 1)
     ```

```
-1
elispy> (- 5 3)
2
elispy> (- 51 4 3 2)
42
```

- `*` … `MulSexpFunction`. Auch hier wieder die ”Spezifikation” des Verhaltens:

  ```
  $ elispy
  elispy> (*)
  1
  elispy> (* 4)
  4
  elispy> (* 4 3)
  12
  elispy> (* 7 3 2)
  42
  ```

- `/` … `DivSexpFunction` Achtung: beachte den Umgang mit 0 Argumenten und mit 1 Argument!

  ```
  $ elispy
  elispy> (/ 2)
  0
  elispy> (/ 6 3)
  2
  elispy> (/ 252 3 2)
  42
  ```

- `<` … `LessThanSexpFunction` Diese sollen entweder `t` oder `nil` zurückliefern und müssen immer zwei Argumente aufweisen:

  ```
  $ elispy
  elispy> (< 3 2)
  nil
  elispy> (< 2 3)
  t
  ```

  Als Argumente sind nur Zahlen erlaubt.

- `<=` … `LessThanOrEqualSexpFunction`

- `=` … `EqualsSexpFunction`

- `>=` … `GreaterThanOrEqualSexpFunction`
- `>` … `GreaterThanSexpFunction`
- `first` … liefert erstes Element der Liste zurück. Zu überprüfen ist die Anzal der Argumente (genau 1) und der der Typ des Arguments (muss eine Liste sein)! Nicht zuerst auf das Auswerten des Arguments vergessen!

```
elispy> (first '(1 2 3))
1
elispy> (first ())
nil
elispy> (first nil)
nil
```

- `rest` … liefert den "Rest" der Liste (also ohne erstes Element) zurück. Überprüfungen analog zu `first`.

```
elispy> (rest ())
nil
elispy> (rest '(1))
nil
elsipy> (rest '(1 2 3))
(2 3)
```

- `cons` … Fügt erstes Argument in zweites Argument (eine Liste) ein; beide werden ausgewertet!

```
elispy> (cons 1 ())
(1)
elispy> (cons 1 '(2 3))
(1 2 3)
elispy> (cons '(1 2) '(3 4))
((1 2) 3 4)
```

- `equal` … überprüft, ob zwei sexps gleich sind (werden vorher ausgewertet, eh klar), oder nicht. Liefert dementsprechend `t` oder `nil` zurück.

```
elispy> (equal 1 1)
t
elispy> (equal 1 ())
nil
```

Dazu müssen wir wissen, wann zwei Sexp gleich sind. Das wiederum führt uns zu dem Problem zurück, dass wir wissen müssen, wann zwei Objekte in C# gleich sind. D.h. wir wollen den Operator `==` überladen. Wenn wir dies tun, dann müssen wir auch den Operator `!=` überladen.

Hmm, und wenn wir den Operator `==` überladen, dann werden wir auch die Methode `Equals` überladen. Und dann müssen wir wiederum die Methode `GetHashCode` überladen.

Wir sehen, hier ensteht ein ganzer Rattenschwanz… Die Beschreibung *wie* diese Methoden zu implementieren sind, ist den Folien zu entnehmen. Hier folgen nur ein paar Tipps:

a) Beginnen wir mit dem Operatorpaar `==` und `!=`, die in der Klasse `Sexp` gemäß den Folien zu implementieren sind.

b) Danach sind gleich die Methoden `Equals(object)` und `Equals(Sexp)` zu implmentieren. Hier ist lediglich das Attribut `is_quoted` zu betrachten, da wir die Position nicht zum Diskriminieren heranziehen. D.h. wir betrachten zwei Sexp als gleich wenn alle Attribute bis auf die Position (so vorhanden) gleich sind.

c) In der Klasse `Sexp` ist weiter auch noch die Methode `GetHashCode` gemäß den Folien zu implementieren. Auch hier wird die Position nicht herangezogen.

d) Jetzt ist es an der Zeit in den Subklassen `SexpAtom` und `SexpList` die Methoden `Equals(object)`, `Equals(Sexp****)` und `GetHashCode` zu überschreiben.

- `setq` … Weist übergebenen Symbol (erstes Argument) einen Wert (zweites Argument). Nur zweites Argument wird ausgewertet (→ setq!) Ergebnis ist Wert der Zuweisung.

- `null` … Erwartet sich genau ein Argument und wertet dieses aus und überprüft, ob es sich einen "Null"-Wert handelt.

  Dafür werden wir die Methode `is_null` in unserer Klassenhierarchie implementieren, die *nur* für die leere Liste `true` zurückliefert, anderenfalls (klarerweise) `false`.

```
elispy> (null 1)
nil
elispy> (null nil)
t
```

- `if` … Erwartet sich genau 2 oder 3 Argumente. Erstes Argument ist die Bedingung und wird ausgewertet und zu einem `bool` konvertiert.

  Dazu ist der Operator `bool` in der Klasse `Sexp` zu implementieren, der genau dann `true` zurückliefert, wenn die Methode `is_null()` `true` zurückliefert:

  ```
  public static explicit operator bool(Sexp sexp) { ... }
  ```

  Zurück zur Implementierung von `if`. Ist der Wert des ersten Argumentes `true`, dann wird das zweite Argument ausgewertet, anderenfalls, ein etwaiges drittes Argument. Das Ergebnis ist das jeweils ausgewertete Argument bzw. `nil`.

  Was aber, wenn im "then" oder im "else" Zweig mehrere Ausdrücke ausgewertet werden sollen? Weiter zur nächsten Funktion.

- `progn` … Erwartet sich beliebig viele Argumente und wertet diese in der gegebenen Reihenfolge aus. Gibt den Wert des letzten Ausdruckes zurück.

  Hier wieder ein Beispiel:

  ```
  elispy> (progn)
  nil
  elispy> (progn 1)
  1
  elispy> (progn 1 (+ 1 1) (- 43 1))
  42
  ```

- `princ` … (Kein Tippfehler!) Erwartet sich genau 1 Argument, das ausgewertet, ausgegeben und zurückgegeben wird. Zu beachten ist die Ausgabe eines Strings und der Wert des Ausdruckes. Beispiel:

  ```
  elispy> (princ 1)
  1
  1
  elispy> (princ "abc")
  abc
  "abc"
  elispy> (princ 'a)
  a
  a
  elispy> (setq a 42)
  42
  elispy> (princ a)
  42
  ```

```
42
elispy> (princ '(1 2 3))
(1 2 3)
(1 2 3)
```

- `while` … Erwartet sich mindestens 2 Argumente. Erstes Argument ist die Bedingung und die weiteren Argumente sind die Sexps, die in der Schleife als Rumpf ausgeführt werden sollen2. Der Wert des `while`-Ausdruckes Ausdruckes ist der Wert der Bedingung.

- `shell` … soll ein Kommando in der bash ausführen und dessen Ausgabe (auf stdout) als String als Ergebnis liefern soll:

  ```
  elispy> (shell "ls")
  ""
  elispy> (shell "ls -a")
  ".
  ..
  "
  ```

  Dazu muss man auch wissen, wie man unter C# einen externen Prozess startet. Ich habe hier eine Version zur Verfügung gestellt, die gerne verwendet werden kann und unter Linux funktioniert (vorausgesetzt die `bash` ist unter `/bin/bash` zu finden):

  ```
  public static string shell_exec(string cmd) {
      // replace " in cmd by \"
      var escaped_args = cmd.Replace("\"", "\\\"");

      var process = new Process() {
          StartInfo = new ProcessStartInfo {
              FileName = "/bin/bash",
              Arguments = $"-c \"{escaped_args}\"",
              RedirectStandardOutput = true,
              UseShellExecute = false,  // false on dotnet core anyway
              CreateNoWindow = true
          }
      };

      process.Start();
      string result = process.StandardOutput.ReadToEnd();
      process.WaitForExit();

      if (process.ExitCode != 0)
  ```

```
            throw new InvalidOperationException($"Process exited with {process.Exit
        return result;
    }
```

- `not` … logisches NICHT. Erwartet sich genau ein Argument, das ausgewertet wird. Jeder "Null"-Wert wird zu `t`, alles andere zu `nil`.

- `and` … logisches UND. Erwartet sich eine beliebige Anzahl von Argumenten. Evaluiert ein Argument nach dem anderen bis ein Ergebnis `nil` wird, dann wird `nil` zurückgeliefert. Werden alle zu nicht `nil` ausgewertet, dann wird der letzte Wert zurückgeliefert:

  ```
  elispy> (and)
  t
  elispy> (and t)
  t
  elispy> (and nil)
  nil
  elispy> (and 1 2 3)
  3
  elispy> (and 1 nil a)
  nil
  ```

  Beachte, dass es zu keinem Fehler kommt, obwohl das Symbol `a` nicht definiert ist!

- `or` … logisches ODER. Erwartet sich eine beliebige Anzahl von Argumenten. Evaluiert ein Argument nach dem anderen bis ein Ergebnis nicht-`nil` wird. Dieses wird dann zurückgeliefert. Werden alle zu `nil` ausgewertet, dann wird `nil` zurückgeliefert:

  ```
  elispy> (or)
  nil
  elispy> (or 1)
  1
  elispy> (or 1 2 3)
  1
  elispy> (or nil nil 3 a)
  3
  ```

5. Eine Kleinigkeit noch, dann sind wir fertig. Wir werden das Programm jetzt so umändern, dass *nicht* mehr der Wert des letzten Ausdruckes auf `stdout` ausgegeben wird. Dies war nur eine Krücke, weil wir noch nicht die Funktion `princ` implementiert hatten! Aber das ist wirklich eine Kleinigkeit.

Teste dies noch indem du ein Programm in `elispy` in der Datei `fact.el` schreibst, das die Faktorielle von 5 berechnet und auf `stdout` ausgibt.

Perfekt!!! Jetzt ist unser Interpreter vollständig funktionsfähig. Das ist ein großer Erfolg.

4.4 Transpiler

Das fulminante Ende, die Codegenerierung!

Implementiere jetzt die Codegenerierung… Hier ein paar Tipps bzw. Anweisungen:

- Teste deine Kommandozeilenverarbeitung jetzt noch speziell unter dem Gesichtspunkt der Verarbeitung der Option `-g`:

  ```
  $ elispy -g
  usage: report [--help|-h|-g] [FILE]
  Evaluate the elispy expressions of FILE otherwise the REPL will be started.

    --help|-h ... Help!
    -g ... generate C# code; only valid if FILE is provided
    FILE ... file name or - (stdin). If FILE is missing start the REPL

  No filename given but code generation requested!
  $ echo "(+ 1 2)" | elispy -g -
  $ elispy -g test.el
  ```

  Die letzten beiden Aufrufe werden anfangs noch keine Aktion nach sich ziehen, aber die Aufrufe sollen korrekt erkannt werden.

- Unter Umständen kannst du bei der Ausgabe über `stdout` folgenden kleinen Trick gut verwenden:

  ```
  StreamWriter sw;
  if (...) {
      sw = new StreamWriter(Console.OpenStandardOutput());
      sw.AutoFlush = true;
  } else {
      sw = new StreamWriter("test.cs");
  }
  sw.WriteLine("xxx");
  ```

  Damit kann man in Abhängigkeit einer Bedingung mittels einer Anweisung die Ausgaben entweder nach `stdout` oder in eine Datei schreiben.

- Hier wieder das obligatorische Interface:

  ```
  public interface CodeGenerator {
      void visit(SexpSymbol sexpsym);
      void visit(SexpString sexpstr);
      void visit(SexpInteger sexpint);
      void visit(SexpList sexplst);
  }
  ```

  Hmm, hier soll das sogenannte *Visitor*-Pattern implementiert werden… Dazu später…

- Implementiere wiederum eine Klasse, die dieses Interface *CodeGenerator* implementiert, nämlich `CSharpGenerator`. Bevor wir uns darüber Gedanken machen *diese* Klasse zu implementieren ist, vorweg noch ein paar Gedanken wie prinzipiell so eine Codegenerierung zu schreiben ist.

  Die grundlegenden Vorgänge zur Codegenerierung sind in den Folien zum Compilerbau angeführt. Allerdings handelt es sich dort um die Grundlagen der Grundlagen der Grundlagen… zum Compilerbau. Wir haben hier ein relativ einfaches, wenn auch nicht triviales Problem zu lösen, nämlich wie für unsere Interpretersprache eine einfache Codegenerierung gelöst werden kann.

  Es gibt prinzipiell zwei Ansätze:

  – Der erste Ansatz basiert darauf, dass man eine vollwertigen Codegenerierung, die von `elispy` nach `C#` umsetzt, implementiert. D.h., dass die Features von C# und auch die nativen Datentypen bestmöglich eingesetzt werden. Damit ist gemeint, dass für den Datentyp `SexpsInteger` unserer Programmiersprache `elispy` auch der Datentyp `int` von `C#` verwendet wird.

    Der Vorteil dieser Variante liegt auf der Hand: Performance! Der Nachteil ist aber auch nicht zu verachten: Komplexität!

  – Der zweite Ansatz basiert darauf, dass man Code generiert, der direkt das Verhalten unseres Interpreters in `C#` nachvollzieht.

    Der Vorteil dieses Ansatzes liegt auch auf der Hand: Komplexität! Aber auch der Nachteil ist klar: Performance!

  Wähle zwischen einer dieser beiden Varianten!

  Ich gebe dir im weiteren ein paar Tipps zur Implementierung des zweiten Ansatzes…

- Zuerst werden wir uns den zu generierenden Code ansehen, wie dieser strukturiert ist und welche prinzipielle Idee dahinter steckt. Erst danach werden wir zur Implementierung schreiten.

  Nehmen wir als einfaches Beispiel folgendes triviales `elispy` Programm in einer Datei `test1.el` her:

```
(setq n 5)
(princ n)
```

  Mittels des Befehls

```
elispy -g test1.el
```

  sollte nachfolgender Code in einer Datei `test1.cs` generiert werden:

```
using System;
using System.Collections.Generic;
using ko.elispy;

class Program {
    public static void Main() {
        var ctx = new Context();
        ctx.symtab["n"] = (new SexpInteger(5)).eval(ctx);
        Console.WriteLine((ctx.symtab["n"]).eval(ctx));
     }
}
```

  Nehmen wir ein weiteres (bekanntes) Beispiel her, indem wir das vorhergehende `elispy` Programm erweitern:

```
(setq n 5)
(setq res (setq res 1))
(while (> n 0) (setq res (* res n)) (setq n (- n 1)))
(princ res)
```

  Dann sollte folgender Code generiert werden:

```
using System;
using System.Collections.Generic;
using ko.elispy;

class Program {
    public static void Main() {
        var ctx = new Context();
        ctx.symtab["n"] = (new SexpInteger(5)).eval(ctx);
        ctx.symtab["res"] = (ctx.symtab["res"] = (new SexpInteger(1)).eval(ctx)).eval(ctx);
        while ((bool)((int)((ctx.symtab["n"]).eval(ctx)) > (int)((new SexpInteger(0)).eval(ctx)).eval(ctx))) {
            (ctx.symtab["res"] = (new SexpInteger(1 * (int)((ctx.symtab["res"]).eval(ctx)) * (int)((ctx.symtab["n"]).eval(ctx)))).eval(ctx)).eval(ctx);
```

```
            (ctx.symtab["n"] = (new SexpInteger((int)((ctx.symtab["n"]).eval(ctx)) - (int)((new SexpInteger(1)).eval(ctx)))).eval(ctx)).eval(ctx);
        };
        Console.WriteLine((ctx.symtab["res"]).eval(ctx));
    }
}
```

Übersetzt und ausgeführt wird/soll es das erwartete Ergebnis zeigen. Versuche die zugrunde liegende Idee zu erfassen und zu verstehen!

- Jetzt zur Implementierung, die auf dem Visitor-Pattern basiert. Dazu werden wir jetzt unsere `Sexp`-Hierarchie um eine weitere Methode implementieren müssen:

  ```
  void accept(CodeGenerator cg);
  ```

  Diese ist entsprechend in der Hierarchie abstrakt bzw. konkret zu implementieren:

  ```
  public override accept(CodeGenerator cg) {
      cg.visit(this);
  }
  ```

  In der Klasse `Elispy` ist beim Generieren des Codes für jeden vollständigen `elispy`-Ausdruck unseres `elispy`-Programmes die Methode `accept` aufzurufen.

  Die gesamte "Intelligenz" steckt in den `visit` Methoden:

  – Für ein `SexpSymbol`, einen `SexpString` und einen `SexpInteger` ist die Sache einfach und entsprechend den Beispielen von vorhin leicht zu implementieren.

    Erweitere dazu deine Klasse `CSharpGenerator` um eine Instanzvariable `code` vom Typ `StringBuilder`, die verwendet wird, um den generierten Code aufzunehmen.

  – Für eine `SexpList` ist dies komplizierter!

    Es sind grundlegend zwei Teile zu realisieren:

    * Erweitere deine Klasse `CSharpGenerator` um Methoden folgender Art:

      ```
      void gen_code(AddSexpFunction func, List<Sexp> args) {
          ...
      }
      ```

      Innerhalb dieser Funktionen wird der Code für die entsprechende Funktion zu `code` hinzugefügt.

    * Weiters ist die Methode `void visit(SexpList sepxlist)` entsprechend zu implementieren.

Diese beiden Teile stellen das Herz unserer Codegenerierung dar.

Der "Template"-Anteil, d.h. das Gerüst eines C#-Programmes kann durch zwei weitere Methoden der Klasse `CSharpGenerator`, nämlich `void start()` und `void end()` leicht generiert werden und ist trivial.

Happy Hacking!

BTW, dieser Ansatz wird oft gewählt, wenn eine Interpretersprache um einen Compiler erweitert werden soll. Ausgehend von diesem Ansatz wird nach um nach der generierte Code optimiert.

5 Übungszweck dieses Beispiels:

- C# lernen!
  - verschachtelte Namensräume verwenden
  - Properties kennenlernen und einsetzen
  - Umgang mit regulären Ausdrücken lernen
  - Implementierung eines `IEnumerable`
    - manuell und
    - auf Basis von `yield return`
  - Einsatz von verbatim string Literalen im speziellen mit doppelten Anführungszeichen
  - Eigene Exceptions definieren und werfen
  - `static` Klassen kennenlernen
  - Konstanten definieren
  - Vererbung von Klassen üben und Konstruktor der Basisklasse aufrufen
  - Methoden mit Default-Parametern definieren
  - generische Methoden implementieren und verwenden
  - Erweiterungsmethoden (extension methods) kennenlernen und implementieren

  - `Nullable<T>` bzw. `?` verwenden
  - Operatoren `is` und `as` einsetzen
  - *using alias directive* einsetzen
  - Eigene Konvertierungsoperatoren implementieren, `explicit` vs `implicit`
  - innere Exception verwenden
  - Implementierung von Klassenhierarchien
  - Überladen von Operatoren
  - Korrektes Implementieren von Operatoren zum Vergleichen auf Gleichheit (`Equals`, `GetHashCode`, `operator==`, `operator!=`)
  - Implementieren eines Operator zum Konvertieren in einen boolschen Wert (`operator bool`)
  - Implementieren eines Operators zum Konvertieren in eine ganze Zahl (`operator int`).
  - Implementieren eines Operators zum Konvertieren in einen String (`operator string`)
  - Implementieren eines Operators zum Konvertieren in ein Objekt eines benutzerdefinierten Typs.
  - Starten von Prozessen in C#
  - Verstehen und Implementieren von Prozessen, die mittels des Pipe-Operators zwischen `stdout` und `stdin` kommunizieren.
  - Erkennen von EOF sowie `CTRL-D` (bzw. `CTRL-Z`) erkennen, verstehen und behandeln.
  - Verwenden von `StringBuilder`
- Üben von regulären Ausdrücken zur Erkennung von Mustern
- Implementierung von Lexern
- Implementierung von einfachen Parsern (recursive-descent parser) und korrektes Aufbauen eines Syntaxbaumes sowie Verwendung einer Symboltabelle.
- Implementierung eines einfachen Interpreters

- Implementierung einer REPL (read-eval-print-loop) Benutzerschnittstelle
- Implementierung einer einfachen Codegenerierung
- Programmiersprachenkonzepte am Beispiel Lisp kennenlernen
- Kennenlernen des Visitor-Pattern (zumindest intuitiv)