BFNP – Functional Programming

Lecture 9: Parsing with F#

Niels Hallenberg

Ray Tracer, Package I - Overview

- Three Dimensional Vectors
- Three Dimensional Points
- Parsing simple Arithmetic Expressions

Points and Vectors

- Ray tracers make use of 3-dimensional *points* and *vectors*.
- The points are used to model points in three dimensional space
- Vectors are used to model directions in this space.

We use tuples of floats to model points and vectors.

```
type Vector = V of float * float * float
```

In the notation below, we write $\{v.x, v.y, v.z\}$ to denote the three values that makes up the vector v.

The same notation is used for points, i.e., $\{p.x, p.y, p.z\}$ for a point p.

We will also use a “dot” notation to access the vector and points components in the formulas below. For instance, $p.x$ is the x component of the point p and $v.z$ is the z component of the vector v.

Three Dimensional Vectors - signature

```
module Vector
[<Sealed>]
type Vector =
  static member ( ~- ) : Vector -> Vector
  static member ( + ) : Vector * Vector -> Vector
  static member ( - ) : Vector * Vector -> Vector
  static member ( * ) : float * Vector -> Vector
  static member ( * ) : Vector * Vector -> float

val mkVector : x:float -> y:float -> z:float -> Vector
val getX : Vector -> float
val getY : Vector -> float
val getZ : Vector -> float
val getCoord: Vector -> float * float * float
val multScalar : Vector -> s:float -> Vector
val magnitude : Vector -> float
val dotProduct : Vector -> Vector -> float
val crossProduct : Vector -> Vector -> Vector
val normalise : Vector -> Vector
val round : Vector -> int -> Vector
```

Three Dimensional Vectors - formulas

- Type `Vector`: An abstract type `Vector` hiding the actual representation of vectors.
- `mkVector` x y z: A function `mkVector` of type `float -> float -> float -> Vector`. The function creates a vector that points from the origin to the point with coordinates x, y, and z.
- `getX` v: A function `getX` of type `Vector -> float`. The function returns the x component of the vector.
- `getY` v: A function `getY` of type `Vector -> float`. The function returns the y component of the vector.
- `getZ` v: A function `getZ` of type `Vector -> float`. The function returns the z component of the vector.
- `multScalar` v s: A function `multScalar` of type `Vector -> float -> Vector`. The function scales the vector v by the float s. The formula is $\{v.x * s, v.y * s, v.z * s\}$.
- ...

Three Dimensional Vectors - operations

- $-$: The unary operator $-v$ has type `Vector -> Vector`. The operator negates the vector v using the formula $\{-v.x, -v.y, -v.z\}$.
- $+$: The binary operator $u + v$ has type `Vector * Vector -> Vector`. The formula is $\{u.x + v.x, u.y + v.y, u.z + v.z\}$.
- $-$: The binary operator $u - v$ has type `Vector * Vector -> Vector`. The formula is $\{u.x - v.x, u.y - v.y, u.z - v.z\}$.
- $\star$: The binary operator $s \star v$ has type `float * Vector -> Vector`. The formula is `multScalar v s` using the function `multScalar` defined above.
- $\star$: The binary operator $u \star v$ has type `Vector * Vector -> float`. The formula is `dotProduct u v` using the function `dotProduct` defined above.

Three Dimensional Vectors - compiling and testing

Generate `Vector.dll`:

```
fsharpc -a Vector.fsi Vector.fs
```

Testing (I assume English comma-setting):

```
$ fsharpc -r Vector.dll VectorTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono VectorTest.exe
VectorTest
Test01 OK
Test02 OK
Test03 OK
Test04 OK
Test05 OK
Test06 OK
Test07 OK
Test08 OK
Test09 OK
Test10 OK
Test11 OK
Test12 OK
Test13 OK
```

Three Dimensional Points - signature

```
module Point
//[<Sealed>]

type Vector = Vector.Vector
type Point

val mkPoint : float -> float -> float -> Point
val getX : Point -> float
val getY : Point -> float
val getZ : Point -> float
val getCoord : Point -> float * float * float
val move : Point -> Vector -> Point
val distance : Point -> Point -> Vector
val direction : Point -> Point -> Vector
val round : Point -> int -> Point
```

Three Dimensional Points - formulas

- Type `Point`: An abstract type `Point` hiding the actual representation of points.
- `mkPoint` x y z: A function `mkPoint` of type `float -> float -> float -> Point`. The function creates a point with the coordinates x, y and z.
- `getX` p: A function `getX` of type `Point -> float`. The function returns the x component of the point.
- `getY` p: A function `getY` of type `Point -> float`. The function returns the y component of the point.
- `getZ` p: A function `getZ` of type `Point -> float`. The function returns the z component of the point.
- `move` p v: A function `move` of type `Point -> Vector -> Point`. The function displaces the point p by the vector v using this formula $\{p.x + v.x, p.y + v.y, p.z + v.z\}$.
- `distance` p q: A function `distance` of type `Point -> Point -> Vector`. The function calculates the distance vector between points p and q. You will obtain a vector that points at q when originating from p. The formual is $\{q.x - p.x, q.y - p.y, q.z - p.z\}$
- ...

Three Dimensional Points - compiling and testing

Generating `Point.dll`:

```
fsharpc -a -r Vector.dll Point.fsi Point.fs
```

Testing (I assume English comma-setting):

```
$ fsharpc -r Point.dll PointTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono PointTest.exe
PointTest
Test01 OK
Test02 OK
Test03 OK
Test04 OK
Test05 OK
Test06 OK
Test07 OK
Test08 OK
Test09 OK
```

Parsing with F# - Overview

- Grammars
- Parsing theory (background information only)
- Parser construction in F#
- Scanners
- Building abstract syntax tree

Parsing - Grammars

Grammar notation:

A *grammar* $G = (T, N, R, \mathtt{S})$ has a set T of terminals, a set N of nonterminals, a set R of rules, and a starting symbol $\mathtt{S} \in N$.

A *rule* has form $\mathtt{A} \;=\; \mathtt{f}_1 \;|\; \ldots \;|\; \mathtt{f}_n$, where $\mathtt{A} \in N$ is a nonterminal, each alternative $\mathtt{f_i}$ is a sequence, and $n \geq 1$.

A *sequence* has form $\mathtt{e}_1 \;\ldots\; \mathtt{e}_m$, where each $\mathtt{e_j}$ is a symbol in $T \cup N$, and $m \geq 0$. When $m = 0$, the sequence is empty and is written Λ.

Simple arithmetic expressions of arbitrary length built from the subtraction operator '−' and the numerals `0` and `1` can be described by the following grammar:

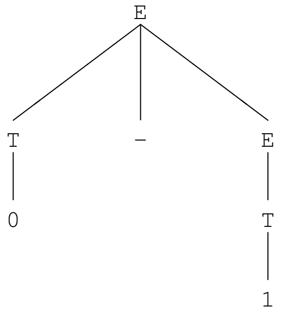
```
E = T "-" E | T .
T = "0" | "1" .
```

The grammar has terminal symbols $T = \{$`"-"`, `"0"`, `"1"`$\}$, nonterminal symbols $N = \{$`E`, `T`$\}$, two rules in R with two alternatives each, and starting symbol `E`. Usually the starting symbol is listed first.

Parsing - Derivation

Example derivation:

```
E ==> T "-" E
  ==> "0" "-" E
  ==> "0" "-" T
  ==> "0" "-" "1"
```



Parsing - Left factorization

The problem is with rules such as `E = T "-" E | T`, where both alternatives start with the same symbol, `T`.

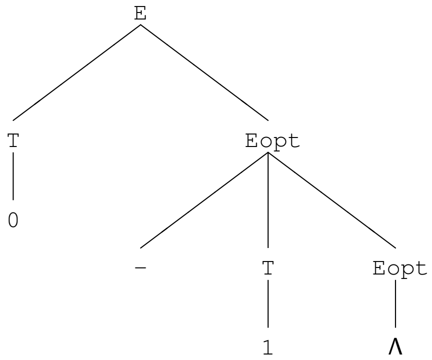
Left factorization of the example above gives:

```
E    = T Eopt .
Eopt = "-" T Eopt | e .
T    = "0" | "1" .
```

where e is the empty sequence Λ.

```
"0" "-" "1"              E
"0" "-" "1"              T   Eopt
"0" "-" "1"              "0" Eopt
    "-" "1"                  "-" T   Eopt
        "1"                      T   Eopt
        "1"                      "1" Eopt
         e                           Eopt
         e                           e
```

Parsing - Left factoring



Parsing - Left recursive nonterminals

There is another type of grammar rules we want to avoid. Consider the grammar

```
E     =  E "-" T | T .
T     =  "0" | "1" .
```

`E` is *left recursive*: there is a derivation `E` $\Longrightarrow$ `E ...` from `E` to a symbol string that begins with `E`.
It is even *self left recursive*: there is an alternative for `E` that begins with `E` itself.
This means that we cannot choose between the alternatives for `E` by looking only at the first input symbol.
Alternative, without left recursion:

```
E     =  T Eopt .
Eopt  =  "-" T Eopt | e
T     =  "0" | "1" .
```

Parsing - construction

The parser, verifying a string against the below grammar

```
E    =  T Eopt .
Eopt =  "-" T Eopt | e
T    =  "0" | "1" .
```

is as follows

```
type terminal = Sub | Zero | One
exception Parseerror

let rec E ts = Eopt(T ts)
and Eopt ts =
  match ts with
    Sub :: tr -> Eopt (T tr)
  | _ -> ts
and T ts =
  match ts with
    Zero :: tr -> tr
  | One  :: tr -> tr
  | _ -> raise Parseerror

let parse ts =
  match E ts with
    [] -> ()
  | _  -> raise Parseerror
```

Parsing - scanners

In the first phase, the character string is converted to a string of terminal symbols, and lay-out information (such as blanks) in the input text is removed. This is called *scanning* or *lexical analysis*
In the second phase, the list of terminal symbols is parsed as described on the previous slide.

```
exception Scanerror
let isblank c = System.Char.IsWhiteSpace c
let explode s = [for c in s -> c]

let scan s =
  let rec sc = function
     [] -> []
   | '-' :: cr  -> Sub  :: sc cr
   | '0' :: cr  -> Zero :: sc cr
   | '1' :: cr  -> One  :: sc cr
   | c   :: cr when isblank c -> sc cr
   | _ -> raise Scanerror
  sc (explode s)

> scan "0-1";;
val it : terminal list = [Zero; Sub; One]
```

Parsing - scanning names

A function to scan names:

```
let isletterdigit c = System.Char.IsLetterOrDigit c
let rec scname (cs, value) =
  match cs with
    c :: cr when isletterdigit c -> scname(cr, value + c.ToString())
  | _ -> (cs, value)
```

A scanner that can also scan names:

```
let isblank c = System.Char.IsWhiteSpace c
let isletter c = System.Char.IsLetter c

type terminal = \ldots | Name of string
let scan s =
  let rec sc cs =
    match cs with
      \dots
    | c :: cr when isblank c -> sc cr
    | c :: cr when isletter c -> let (cs1, n) = scname(cr, (string)c)
                                 Name n :: sc cs1
    | _ -> raise Scanerror
  sc (explode s)
```

Parsing - scanning integer and floating values

We use "." to decide whether we return an integer or a float.

```
let floatval (c:char) = float((int)c - (int)'0')
let intval(c:char) = (int)c - (int)'0'
type terminal =
  Add | ... | Int of int | Float of float

let rec scnum (cs, value) =
  match cs with
    '.' :: c :: cr when isdigit c -> scfrac(c::cr,(float)value,0.1)
  | c :: cr when isdigit c -> scnum(cr, 10* value + intval c)
  | _ -> (cs,Int value) (* Number without fraction is an integer. *)
and scfrac (cs, value, wt) =
  match cs with
    c :: cr when isdigit c -> scfrac(cr, value+wt*floatval c, wt/10.0)
  | _ -> (cs, Float value)
```

Parsing - building abstract syntax tree

An *abstract syntax tree* is a representation of a text which shows the structure of the text and leaves out irrelevant information, such as the number of blanks between symbols.
Example abstract syntax tree:

```
type expr = Zeroterm
          | Oneterm
          | Minus of expr * expr
```

The declaration says: an expression is a zero, or a one, or an expression minus another expression.
Below parser builds abstract syntax trees for simple arithmetic expressions.

```
let rec E ts = Eopt (T ts)
and Eopt (ts, inval) =
  match ts with
    Sub :: tr -> let (ts1, tv) = T tr
                 Eopt (ts1, Minus(inval, tv))
  | _ -> (ts, inval)
and T ts =
  match ts with
    Zero :: tr -> (tr, Zeroterm)
  | One :: tr -> (tr, Oneterm)
  | _ -> raise Parseerror
```

Parsing - building abstract syntax tree

The new parsing functions have types

```
E     : terminal list        -> terminal list * expr
Eopt  : terminal list * expr -> terminal list * expr
T     : terminal list        -> terminal list * expr
parse : terminal list        -> expr
```

Typical uses of the new parser are

```
> parse [One;Sub;Zero];;
val it : expr = Minus (Oneterm,Zeroterm)

> parse (scan "0-1-1");;
val it : expr = Minus (Minus (Zeroterm,Oneterm),Oneterm)
```

Parsing simple Arithemetic Expessions - Overview

- Grammar for Ray Tracer expressions
- Scanner for simple gammar
- Syntactic sugar
- Parser for simple grammar
- Visualizing abstract syntax with Graphviz

Parsing - Grammar for Ray Tracer expressions

The grammar

```
E     = E "+" E
      | E "*" E
      | E "^" Int
      | Int
      | Float
      | Var
      | "(" E ")" .
```

becomes

```
E     = T Eopt .
Eopt  = "+" T Eopt | e .
T     = F Topt .
Topt  = "*" F Topt | e .
F     = P Fopt .
Fopt  = "^" Int | e .
P     = Int [ Float | Var | "(" E ")" .
```

Scanner for simple grammar

```
let scan s =
  let rec sc cs =
    match cs with
      [] -> []
    | '+' :: cr -> Add :: sc cr
    | '*' :: cr -> Mul :: sc cr
    | '^' :: cr -> Pwr :: sc cr
    | '(' :: cr -> Lpar :: sc cr
    | ')' :: cr -> Rpar :: sc cr
    | '-' :: c :: cr when isdigit c->let (cs1,t)=scnum(cr,-1*intval c)
                                     t :: sc cs1
    | c :: cr when isdigit c -> let (cs1, t) = scnum(cr, intval c)
                                t :: sc cs1
    | c :: cr when isblank c -> sc cr
    | c :: cr when isletter c -> let (cs1, n) = scname(cr, (string)c)
                                 Var n :: sc cs1
    | _ -> raise Scanerror
  sc (explode s)
```

Scanner for simple grammar

Examples:

```
> scan "2x(2x)";;
val it : terminal list = [Int 2; Var "x"; Lpar; Int 2;
                           Var "x"; Rpar]
> scan "2*x*(2*x)";;
val it : terminal list =
  [Int 2; Mul; Var "x"; Mul; Lpar; Int 2; Mul;
   Var "x"; Rpar]
>
```

Notice that the first string `"2x(2x)"` has implicit multiplications. The second example have all multiplications inserted and the resulting list of terminals fulfils the grammar.
We would like to allow writing the expressions with implicit multiplications `"*''`.

Syntactic sugar - `insertMult`

The task is to implement a function `insertMult` *`ts`* where *`ts`* is a list of terminals. The function returns a new list of terminals where implicit multiplications are inserted explicitly.

```
let rec insertMult = function
  Float r :: Var x :: ts -> [] // TO DO
| Float r1 :: Float r2 :: ts -> [] // TO DO
| Float r :: Int i :: ts -> [] // TO DO
| ...
| Float r :: Lpar :: ts -> [] // TO DO
| Var x :: Lpar :: ts -> [] // TO DO
| Int i :: Lpar :: ts -> [] // TO DO
| t :: ts -> t :: insertMult ts
| [] -> []
```

Example:

```
scan "2 x y z";;
val it : terminal list = [Int 2; Var "x"; Var "y"; Var "z"]
> insertMult [Int 2; Var "x"; Var "y"; Var "z"];;
val it : terminal list = [Int 2; Mul; Var "x"; Mul;
                          Var "y"; Mul; Var "z"]
>
```

Syntactic sugar - `insertMult` - compiling and testing

You compile

```
$ fsharpc ExprParse.fs ExprParseTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
```

and execute the unit tests.

```
$ mono ExprParseTest.exe
ExprParseTest
TestScan01 OK
...
TestInsertMult01 OK
TestInsertMult02 OK
TestInsertMult03 OK
...
$
```

Parser for simple grammar

Abstract syntax to represent simple expressions:

```
type expr =
  | FNum of float
  | FVar of string
  | FAdd of expr * expr
  | FMult of expr * expr
  | FExponent of expr * int
```

A few examples:

```
>parse(insertMult(scan "2 x y z"));;
val it : expr = FMult (FMult (FMult (FNum 2.0,FVar "x"),
                             FVar "y"),FVar "z")
> parse(insertMult(scan "2 x^2"));;
val it : expr = FMult (FNum 2.0,FExponent (FVar "x",2))
>
```

Parser for simple grammar - signature

```
module ExprParse
[<Sealed>]

type terminal
exception Scanerror
val scan: char seq -> terminal list
val insertMult: terminal list -> terminal list

type expr
exception Parseerror
val parse: terminal list -> expr
val dotAST: expr -> string
```

Parser for simple grammar - template

```
let rec E (ts:terminal list) = (T >> Eopt) ts
and Eopt (ts, inval) =
  match ts with
    Add :: tr -> ...
and T ts = ...
and Topt (ts, inval) = ...
and F ts = ...
and Fopt ts = ...
and P ts = ...

let parse ts =
  match E ts with
    ([], result) -> result
  | _ -> raise Parseerror
```

Parsing - Visualizing abstract syntax with Graphviz

The function `dotAST` can generate a `.dot`[1] file.
For instance:

```
> dotAST (parse (insertMult (scan "2x(2x)")));;
```

generates the following text string:

```
val it : string =
  "digraph G {
label="FMult (FMult (FNum 2.0,FVar "x"),FMult (FNum 2.0,FVar "x"))
"Node2 [label="2"];
Node3 [label="x"];
Node4 [label="*"];
Node6 [label="2"];
Node7 [label="x"];
Node8 [label="*"];
Node9 [label="*"];
Node4 -> Node2;
Node4 -> Node3;
Node8 -> Node6;
Node8 -> Node7;
Node9 -> Node4;
Node9 -> Node8;}"
>
```

[1] A `.dot` is a file format for specifying graphs. The application `Graphviz` can show such graphs, see `http://www.graphviz.org/doc/info/lang.html`

Parsing - Visualizing abstract syntax with Graphviz

Copying the text into a file, say `ast.dot` you can generate graphical picture of the AST:

```
$ dot -Tpng ast.dot -o ast.png
```

generates the file `ast.png`.