Laboration #3: MFL – Minimal Functional Languge

Deadline: 2015-05-26

I den här laborationen är din uppgift att konstruera en interpretator för ett enkelt funktionellt programspråk, kallat MFL (Minimal Functional Language). Laborationen är utformad för att uppfylla följande syften:

- Ge studenten övning i att arbeta med samlingar, träd och funktionspekare.
- Ge studenten bättre känsla för hur programspråk fungerar.
- Ge studenten tillfälle att implementera en *rekursivt nedstigande parser*.

Vad är MFL?

MFL (Minimal Functional Language) är ett påhittat minimalt funktionellt språk. Språket stöder heltalsberäkningar med de fyra räknesätten samt funktioner med ett argument. I MFL är allt uttryck, även funktioner så dessa kan skickas som argument m.m.

I laboration skapas en interpretator för MFL som liknar en Haskell-interpretator i sin funktion, d.v.s. interpretatorn visar en prompt där användaren kan skriva ett uttryck som interpretatorn sedan beräknar värdet på. Interpretatorn skall också stödja ett antal kommandon, t.ex. för att lägga till eller ladda in definitioner.

Kommandon i MFL interpretatorn

Följande avsnitt är en förteckning över samtliga kommandon som interpretatorn för MFL känner/skall känna till.

`:load <file>` Kommandot laddar in definitionerna i filen <file>.

`:define <id> = <expression>` Definiera <id> som uttrycket <expression>.

`:help` Kommandot skriver ut ett enkelt hjälpmeddelande som beskriver hur interpretatorn fungerar.

`:quit` Avslutar MFL interpretatorn. Det enklaste sättet att implementera detta kommando är att anropa funktionen **exit(0)**.

Varje kommando skall kunna förkortas till sin första bokstav, t.ex. **:q** för att avsluta.

Sammanfattning av uttryck i MFL

Först och främst så är alla typer av heltalsuttryck med **+**, **-**, *****, **/** och parenteser med heltalskonstanter giltiga uttryck i MFL. Utöver detta finns också if-then-else uttryck och funktionsuttryck (som skapar en funktion) och funktionsanrop (som anropar en funktion). Ett smakprov på MFL definitioner i en fil och en interaktiv session med interpretatorn följer nedan:

Laboration #3: MFL – Minimal Functional Languge

```
# File: test.mfl
# Example definitions
X = 42
zero = 0
id = func (x) { x }
fac = func (n) {
     if n = 0 then 1
     else n*fac(n-1)
}
Bot = bot
```

```
MFL interpreter (type ":help" for help)
> 1+3*4
13
> :load test.mfl
> x
42
> fac(4)
24
> fac
<function with one argument>
> func (x) { 2*x }
<function with one argument>
> func (x) { 2*x }(3)
6
> bot
Error: Stack overflow. Too deep recursion.
>
```

Notera hur namn ges till uttryck i filen medan den interaktiva sessionen evaluerar uttryck. I en definitionsfil kan definitioner löpa över flera rader så länge fortsättningen inte börjar i den första kolumnen. Rader som börjar med ’#’ är kommentarer som ignoreras vid inladdning av filen.

MFL-interpretatorn

Det är nu dags att titta närmare på exakt vad den här laborationen går ut på samt vad som skall konstrueras.

Din uppgift är att skriva en interpretator för det funktionella språket Minimal Functional Language (MFL). En interpretator är ett program som läser in uttryck och kommandon från användaren och agerar baserat på dem.

```
while (TRUE)
{
     read command from user
     act on command
}
```

Interpretatorns struktur är således relativt enkel. I kursboken beskrivs en interpretator som enbart evaluerar aritmetiska uttryck men i det här fallet är uttrycken från språket MFL. Om användaren skriver in ett uttryck så skall interpretatorn beräkna uttryckets värde (d.v.s. evaluera det) och skriva ut detta värde. Om användaren skriver in kommandot **`:define`** så skapas en ny definition. Om användaren skriver in kommandot **`:quit`** så avslutas interpretatorn.

Några detaljer

Det är nu dags att diskutera tillvägagångssättet för konstruktionen av interpretatorn. De huvudsakliga detaljerna rör parsning och evaluering av uttryck i MFL då hela MFL-språket utgörs av uttryck. Parsning utgår från språkets grammatik (syntax) medan evaluering utgår ifrån språkets semantik (d.v.s. vad konstruktioner i språket betyder / hur de beräknas till värden). Båda problemen kommer beskrivas i detalj nedan.

Laboration #3: MFL – Minimal Functional Languge

Tekniken du skall använda liknar mycket den som diskuteras i kapitel 14 i kursboken. I boken används en *rekursivt nedstigande parser* som analyserar ett uttryck och skapar en **`expADT`** som representerar uttrycket. För att evaluera ett uttryck används en evalueringsfunktion som tar en **`expADT`** och avgör dess värde. Den huvudsakliga skillnaden mellan interpretatorn i kursboken och den här interpretatorn är att här används en annan mer omfattande grammatik. Det gör också att evalueraren blir mer omfattande än den i kursboken.

Parsning av uttryck

Rekursivt nedstigande parsning beskrivs i kursboken och har diskuterats på föreläsning 9. Idén är att uttryck beskrivs av en uppsättning regler som kallas för en grammatik och parsern för motsvarande grammatik består av en mängd ömsesidigt rekursiva funktioner. En rekursiv funktion för varje grammatisk konstruktion.

En viktig detalj vid konstruktion av grammatiker och parsers är hur operatorernas precedens hanteras. Kursboken beskriver en enkel ansats, *Iversonian precedence*, som är enkel att implementera men som inte är korrekt för språk som MFL. Kursboken beskriver vidare en teknik (se koden på sidan 621) där precedensen är inbyggd i parsern. Olyckligtvis gör tekniken att parsern blir relativt invecklad. I stället för att använda en enkel rekursivt nedstigande parsningsteknik måste parsern veta något om vilken typ av resultat vi vill ha. Det är givetvis en godtagbar ansats som ofta beskrivs i böcker om parsning men i den här laborationen skall vi använda en annan ansats som bygger på rekursivt nedstigande parsning.

Då vi i den här laborationen använder en rekursivt nedstigande parser kommer *operatorernas precedens att byggas in i grammatiken* istället för i parsern. Vi skall alltså använda en grammatik som ger oss en otvetydig parsning av uttryck och som på ett korrekt sätt avspeglar operatorernas precedens. Här är den grammatik för MFL som vi skall använda:

```
Def ::= identifier = E                  // Only in files and in the
                                        // :define command
E ::= T + E
E ::= T - E
E ::= T

T ::= C * T
T ::= C / T
T ::= C

C ::= F (E)                             // Call to function.
C ::= F

F ::= (E)
F ::= if E RelOp E then E else E        // If expression
F ::= func (identifier) { E }           // Function with argument
F ::= integer
F ::= identifier

RelOp ::= =
RelOp ::= <
RelOp ::= >
```

Precis som i boken skall du skriva funktioner, **`ReadE`**, **`ReadT`**, **`ReadC`** och **`ReadF`** som motsvarar de grammatiska konstruktionerna. Samtliga funktioner returnerar en **`expADT`**.

Laboration #3: MFL – Minimal Functional Languge

Då varje grammatisk konstruktion har fler än en regel är en uppgift som funktionerna måste utföra att avgöra vilken regel som skall appliceras. Det är lite svårare här än för grammatiken i kursboken. För att illustrera hur funktionerna skall fungera beakta hur **`ReadT`** måste fungera vilket beskrivs nedan:

- Anropa **`ReadT`**. Om ingen T existerar anropas funktionen **`Error`**. Om en **`C`** existerar sparas motsvarande **`expADT`**.

- Kontrollera om det finns någonting kvar i scannern. Om inte, returnera den sparade **`expADT`** för **`C`**.

- Om något finns kvar i scannern läs in nästa token från scannern och avgör om det är **`"*"`** eller **`"/"`**. Om inte, lägg tillbaka inläst token till scannern och returnera den sparade **`expADT`** för **`C`**. Anropande funktion tar hand om det som återstår i scannern.

- Om **`"*"`** eller **`"/"`** återfanns, anropa **`ReadT`** rekursivt för att läsa denna, bygg upp motsvarande **`expADT`** med hjälp av **`T`** och den sparade **C** och returnera detta.

Med detta utkast, och koden i kursboken som utgångspunkt, skall det vara möjligt att skriva funktionerna **`ReadE`**, **`ReadT`**, **`ReadC`** och **`ReadF`** på samma sätt (som klienter till **`expADT`**). Notera att parserfunktionerna skall vara noggranna, t.ex. när ett nyckelord förväntas skall detta token kontrolleras och inte bara hoppas över i scannern.

Då funktionerna ovan är implementerade bör du skriva en wrapperfunktion som **`ParseExp`** på sidan 619. Funktionen **`ParseExp`** har som uppgift att läsa in ett helt MFL-uttryck och se till att det inte finns några tokens kvar i scannern då **`ReadE`** returnerar (för i så fall så föreligger ett syntaxfel). Funktionen **`ParseExp`** returnerar en **`expADT`** som representerar det uttryck som parsats.

En intressant detalj att notera med grammatiken är att den är högerassociativ istället för vänsterassociativ! Det är betydligt svårare att konstruera en grammatik som är vänsterassociativ och det är inte värt besväret för den här laborationen. Det här är anledningen till att parserbaserade precedenstekniken används i kursboken; för att åstadkomma önskat vänsterassociativt beteende utan att behöva hantera en alltför komplex grammatik. Vi gör motsatt kompromiss. Genom att acceptera ett högerassociativt beteende så kan du få erfarenhet av att konstruera en rekursivt nedstigande parser.

Vad får detta för implikationer för språket vi implementerar? Här är ett exempel:

I C är semantiken för uttrycket 5 – 4 – 3 följande (5 – 4) – 3. Parenteser krävs för att ändra från vänsterassociativt till högerassociativt beteende.

Med grammatiken för MFL är semantiken av uttrycket 5 – 4 – 3 följande 5 – (4 – 3). Parenteser krävs för att ändra från högerassociativitet till vänsterassociativitet, (5 – 4) – 3. Vi accepterar den här avvikelsen från den normala för att förenkla grammatiken och låta dig använda en enkel rekursivt nedstigande teknik.

Laboration #3: MFL – Minimal Functional Languge

Notera att i båda fallen är 5 + 4 * 3 ekvivalent med 5 + (4 * 3) och 5 * 4 + 3 är ekvivalent med (5 * 4) + 3. Operatorprecedensen är alltså inbyggd i grammatiken.

Evaluering av uttryck : MFL-språkets semantik

Det vi vill göra med uttryck är att evaluera dem. Det första steget är att parsa uttrycken (se ovan) och sedan använda en funktion som liknar **`EvalExp`**, och dess hjälpfunktioner, i kursboken för att beräkna uttryckens värde. Vad som händer när ett uttryck evalueras bestäms av programspråkets semantik (d.v.s. vad konstruktioner i språket betyder). I MFL så blir resultatet av att evaluera/beräkna ett uttryck antingen ett heltalsvärde eller en funktion. Om man man vill beskriva MFL i programspråkstermer är språket rent funktionellt (purely functional) och lat (lazy). Ett uttryck i MFL evalueras i en viss miljö som bestämmer vilka definitioner som finns tillgängliga. Därför blir prototypen för evalueringsfunktionen **`Eval`** som följer:

```
valueADT Eval(expADT exp, environmentADT env);
```

Heltalskonstant-uttryck

Ett heltalskonstant-uttryck evalueras till motsvarande heltalsvärde. Notera att värdena i MFL hanteras som den abstrakta datatypen **`valueADT`** eftersom både heltal och funktioner är värden.

Operator-uttryck

Semantiken för operatoruttryck är som man kan förvänta sig och kan evalueras som i kursbokens exempel. Om höger eller vänsterledet inte blir ett heltalsvärde så är detta ett runtime-fel som signaleras via **`Error`** funktionen.

Identifierar-uttryck

En identifierare i MFL definierar ett uttryck så semantiken är att identifieraren beräknas till värdet på detta uttryck. Detta är dock mer komplicerat att implementera än vad det först verkar eftersom MFL tillåter oss att definiera uttryck vars värde det inte går att beräkna (t.ex. definitionen **`bot = bot`**). Därför går det alltså inte att spara en definition som ett (namn, evaluerat värde)-par i en symboltabell.

Vidare så är evalueringen av identifierarens uttryck beroende av miljön/kontexten där definitionen gjordes p.g.a. att uttrycket kan använda andra identifierare som var synliga där definitionen gjordes. T.ex. så är alla definitioner som finns i en fil tillgängliga för/möjliga att använda i definitionerna i filen.

```
# File: test.mfl
x = 42 + 0
# y is defined in terms of x.
y = 2*x
# Function using yin an inner scope where x
# is shadowed.
f = func (x) { x + y }
# When f(2) is evaluated y should still be 2*42
```

Värdet på identifierarens uttryck skall emellertid inte vara beroende av platsen där identifieraren uppträder. I koden ovan så är värdet på y fortfarande 84 även när y förekommer

inuti funktionen f där det globala x är överskuggat av argumentet x. Detta gör att vi inte bara kan spara uttrycket för en identifierare som ett löst uttryck (expADT) utan vi måste också spara en pekare till miljön där identifieraren definierades. Problemet kan (och skall) hanteras genom att istället associera identifieraren med ett funktionsvärde (se funktions-uttryck och funktionsanropsuttryck nedan) som har uttrycket som funktionskropp och kan anropas när identifierarens värde behöver beräknas. Funktionsvärdet kommer inte ha något formellt argument (om definitionens uttryck är ett funktionsuttryck så kommer evalueringen av identifierarens funktionsvärde resultera i funktionsvärdet för dess uttryck) utan man sätter lämpligen det formella argumentet till NULL. I MFL-interpretatorn kommer definitioner lagras i den ogenomskinliga datatypen environmentADT.

If-uttryck

Semantiken för ett if-uttryck är att först evalueras båda sidorna av villkoret och sedan den gren (then eller else) som blir aktuell att evaluera utifrån värdet på villkoret. Jämförelseoperatorerna <, > eller = ingår inte direkt i MFL-uttryck utan kan bara förekomma i villkoret i if-uttryck. De hanteras därför genom att förlita sig på det faktum att efter nyckelordet if måste det finnas två MFL-uttryck separerade med en jämförelseoperator. Det är möjligt att parsa dessa uttryck och sedan använda evalueraren för att evaluera dem för att avgöra vilken gren av if -uttrycket skall evalueras. Om endera grenen av if-uttrycket inte kan evalueras till ett heltalsvärde (d.vs. om det blir ett funktionsvärde) så är detta ett runtime-fel som signaleras via Error funktionen. Det är inte nödvändigtvis så att båda grenarna av if-uttrycket ger ett värde av samma typ.

Funktions-uttryck

Semantiken för ett funktions-uttryck i MFL är att det skapar ett funktionsvärde som sedan kan anropas (genom ett funktionsanrops uttryck, se nedan). Funktions-uttrycket definierar

```
# File: test.mfl
# Crude way to make a function with two arguments.
add = func (x) { func (y) { x + y } }
# Call: (add(2))(4) # returns 6
# The function above returns a new function that
# has captured x in its closure.

# This can also be used in definitions
firstInc = add(0)
secondInc = add(8)
# Test cases:
# Call: firstInc(3)      # returns 3
# Call: secondInc(3)     # returns 11
# Call: firstInc(1)      # returns 1
# Call: secondInc(2)     # returns 10
```

funktionens formella argument och dess kropp. Funktionsvärdet som skapas kommer också att spara det formella argumentets namn och uttrycket (utan att beräkna det) som utgör funktionens kropp. Precis som för definitioner så måste identifierare som används i funktionens kropp referera till de som var synliga där funktionsuttrycket beräknades. Därför måste funktionsvärdet också innehålla en pekare till den miljö som var synlig där funktionsuttrycket

Laboration #3: MFL – Minimal Functional Languge

beräknades. Det är värt att notera att eftersom funktionsuttryck kan förekomma inuti andra funktioner så kommer miljön/kontexten som bestämmer vilka definitioner som syns utgöras av ett antal lager (*eng.* nested lexical scope) där uppslagning av en identifierare skall söka inifrån och ut (se vidare under funktionsanrops-uttryck). Detta måste implementationen av environmentADT ta hänsyn till (se env.c). Hela det anropningsbara funktionsvärdet (funktion + miljön) kallas för *closure* på engelska.

Funktionsanrops-uttryck
Semantiken för ett funktionsanrops-uttryck är att det beräknas till resultatet av funktionsanropet. För att beräkna värdet på ett funktionsanrop måste uttrycket i funktionens kropp evalueras i en miljö där det formella argumentet (en identifierare) är satt till det faktiska argumentet i anropet. För att kunna göra det måste först funktionsvärdet som anropet görs på beräknas.

Sedan måste en ny miljö som innehåller de definitioner som var synliga när funktionskroppen definierades skapas samt funktionens formella argument definieras som uttrycket för det faktiska argumentet i denna. Att det formella argumentet definieras i miljön gör att en ny miljö måste skapas - antingen här eller när funktionsvärdet skapas så att inte andra användare av originalmiljön påverkas. Notera också att miljön som det faktiska argumentet skall evalueras i (när/om det behövs) är den där anropet sker, vilket man får ta hänsyn till när det formella argumentet definieras. Därefter kan funktionsanropet evalueras genom att beräkna uttrycket i funktionens kropp i den nya miljön.

Om ett funktionsanrop görs på ett värde som inte är ett funktionsvärde så är detta ett runtime-fel som signaleras via Error funktionen. Det är också ett fel om en funktion med argument anropas utan argument eller tvärt om – kom ihåg att namngivna uttryck sparas som argumentlösa funktionsvärden.

Att komma igång
I kurskatalogen finns en del kod som med fördel kan användas vid genomförandet av laborationen. Följande moduler för MFL interpretatorn finns med och bör användas:

```
exp.h,c              /* Abstract syntax ADT for MFL */
value.h,c            /* Value ADT for MFL */
env.h,c              /* Environment/context ADT for MFL */
```

Utöver detta finns gränssnitten för modulerna **`parser.h`**, **`eval.h`** och **`print.h`** och följande användbara moduler:

```
scanadt.h,c
symtab.h,c
```

Där finns även en kompilerad exempelversion av en MFL interpretator som ni kan testa.
I kurskatalogen finns även ett underbibliotek med källkoden för bokens interpretator för heltalsuttryck, och en kompilerad version av motsvarande program. Programmet är en interpretator för aritmetiska uttryck precis som den som beskrivs i kursboken. Programmet hanterar inte BASIC men delas ut därför att den illustrerar hur ni skall hantera en mycket viktig detalj: undantagshantering.

Laboration #3: MFL – Minimal Functional Languge

Undantagshantering

En mycket viktig del i en interpretator är felhantering. Om användaren skriver in ett program och råkar skriva ett syntaxfel så är det inte önskvärt att hela interpretatorn kraschar. Det är dock väldigt bekvämt att anropa funktionen **`Error`** i **`genlib`** för att producera felmeddelanden. Det bästa sättet att hantera felen är att låta interpretatorn känna av och återhämta sig från anrop till **`Error`** med en teknik som kallas för undantagshantering (*exception handling*).

Detaljerna för hur undantagshantering fungerar är inte fokus för den här laborationen. Koden som du behöver finns redan implementerad i **`interp.c`** i interpretatorn för heltalsuttryck och ser ut som nedan:

```
main()
{
    scannerADT scanner;
    expressionADT exp;
    string line;
    InitVariableTable();
    scanner = NewScanner();
    SetScannerSpaceOption(scanner, IgnoreSpaces);
    while (TRUE) {
        try {
            printf("=> ");
            line = GetLine();
            if (StringEqual(line, "quit")) exit(0);
            SetScannerString(scanner, line);
            exp = ParseExp(scanner);
            value = EvalExp(exp);
            printf("%d\n", value);
            except(ErrorException)
            printf("Error: %s\n", (string) GetExceptionValue());
        } endtry
    }
}
```

Det nya i den här koden är **`try`**-satsen vilken är definierad i gränssnittet **`exception.h`** (är en del av biblioteket till kursboken så det är bara att inkludera). **`Try`**-satsen fungerar så att den utför satserna i dess kropp vilka utgörs av satserna fram till **`except`**-satsen. Om inga fel inträffar exekveras samtliga satser i **`try`**-blocket varpå **`while`**-loopen exekveras ytterligare en gång. Om funktionen **`Error`** anropas någon gång under exekveringen av **`try`**-blocket (oavsett hur djupt nästlat anropet till **`Error`** är) så återgår kontrollen omedelbart till **`except`**-satsen varpå felmeddelande skrivs ut. Då **`try`**-satsen har hanterat undantaget avslutas inte programmet som det brukar göra vid anrop till **`Error`**. Programmet fortsätter istället efter **`try`**-satsen och läser in nästa kommando.

Interpretatorn är i stort sett oanvändbar utan den här felhanteringen. Interpretatorn skall ha ett **`try`**…**`endtry`**-block som i exemplet ovan. Det blir dock nödvändigt att förändra detaljerna i själva blocket för att passa interpretatorn för MFL.

Laboration #3: MFL – Minimal Functional Languge

Krav

Din uppgift i den här laborationen är att skriva en MFL-interpretator som den beskrivs i den här laborationsspecifikationen. Då du kodar programmet skall du se till att din lösning uppfyller följande krav:

1. *Programmet skall vara av samma goda modulära struktur som programmet i kursboken som evaluerar uttryck.* Förslag på moduler: huvud-, abstrakt syntax-, parser-, värde-, evaluator-, och utskriftsmodul (för uttryck och värden, det första för debugging och det senare för att skriva ut resultat).

2. *Interpretatorns kommandon skall implementeras med hjälp av en command table som det beskrivs i avsnitt 11.7 i kursboken*. Med andra ord, då ditt program träffar på ett kommando som t.ex. **`:define`** eller **`:load`** skall programmet anropa den funktion som motsvarar kommandot genom att slå upp kommandot i en symboltabell och anropa den funktionspekare som är associerad med kommandot. Samtliga kommandon skall anropas via samma symboltabell. Notera att funktionsprototypen för de funktioner som sparas i symboltabellen måste vara samma, därför kommer eventuellt vissa funktioner för kommandon inte att använda någon eller samtliga argument i prototypen.

3. *Du skall skriva en egen parser enligt beskrivningen ovan.*

4. *Du skall skriva en evalueringsfunktion enligt beskrivningen ovan.*

5. *Ditt program måste ha en rimlig felhantering.* Då det är möjligt att var som helst i programmet anropa funktionen **`Error`** för att åstadkomma ett återhopp till den yttre interpretator-loopen är felhantering relativt enkel. Ditt program måste åtminstone hantera vanligt förkommande fel, som t.ex. syntaxfel i uttryck eller kommandon, evaluering av en odefinierade definition, evaluering av heltalsoperator med en operand som inte är (evalueras till) ett heltal, funktionsanrop på ett värde som inte är ett funktionsvärde, heltalsdivision med 0. Programmet bör också stoppa potentiellt oändliga (läs: för stora) beräkningar innan dessa kraschar hela programmet.

Tips

1. Att testa och felsöka interpretatorn kan vara mycket frustrerande då det är mycket som behöver vara på plats för att kunna börja testa den. Det är lämpligt att börja med utskrift funktionen för uttryck (och värden) för att sedan fortsätta med parsern. Vänta med evalueringen tills det går att parsa uttryck.

2. Starta tidigt! Den här laborationen är relativt stor och komplex så vänta inte med att påbörja den till dagen före deadline. Se även till att vara väl förberedd inför handledningarna.

Att lämna in

Till den här laborationen skall samtliga filer, .c och .h, som du har skapat eller modifierat lämnas in via PingPong. Om ni jobbar i Visual Studio går det bra att skicka ett zip-arkiv med projektbiblioteket.