Domänspecifikt språk för konfiguration i multipla miljöer

DANIEL HENELL
och HENRIK MATTSSON

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Domänspecifikt språk för konfiguration i multipla miljöer

D A N I E L H E N E L L
o c h H E N R I K M A T T S S O N

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Mads Dam
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/henell_daniel_OCH_mattsson_henrik_K10044.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

This article describes the design and implementation of a domain specific language for configuration of multiple runtime environments developed as a case study of the use of the Haskell monadic parser combinator library Parsec for domain specific language implementation.

Sammanfattning

Denna uppsats beskriver design och implementation av ett domänspecifikt språk byggt för att hantera konfiguration av system i multipla driftmiljöer. Den är skriven som en fallstudie i bruket av Haskell tillsammans med det monadiska parserkombinatorbiblioteket Parsec för implementation av domänspecifika språk.

Innehåll

Kapitel 1

Introduktion

1.1 Problem

Ett system kan behöva köras i ett flertal olika miljöer, exempelvis utvecklingsmiljö, testmiljö, drifttestmiljö och i skarp drift. I större system finns ofta en mängd parametrar som ställs in i konfigurationsfiler.

Vanligtvis finns det parametrar som skiljer sig åt mellan de olika miljöerna. Till exempel bör inte samma databas användas vid test likväl som vid skarp drift. Då parametrarna varierar mellan målmiljöer finns behov av olika konfigurationer för olika miljöer.

Mindre skillnader i konfiguration leder då till dubbelarbete när inställningar behöver förändras globalt över samtliga miljöer. Ett större problem kan dessutom vara att förändringar i konfigurationen inte propageras till samtliga målmiljöer vilket kan orsaka svårlösta buggar vid driftsättning.

1.1.1 Existerande lösningar

Skräddarsydda lösningar byggda på macrospråk som M4 eller CPP medför ett preprocessorsteg mellan modifikation och driftsättning. Injektion av lokala värden som i fallet Apache Ant[ant] är också ett existerande alternativ.

Mer eller mindre hårdkodade variabelkontroller som exempelvis i Caucho Resin[res], där en och samma konfigurationsfil kan användas både i GPL- och betalversionerna av applikationsservern existerar också.

I Visual Studio 2010 används XDT (XML Document Transform) för att manipulera en allmän XML-konfigurationsfil för att anpassa den till olika miljöer. Varje miljö har då en XDT-fil med just de förändringarna som behövs för att gå från den allmänna konfigurationen till att passa för den specifika miljön. Parametrar som inte ändras behåller då sitt standardvärde och på så sätt behöver parametrar inte dupliceras i flera fristående konfigurationsfiler.

1.1.2 Exekverbara konfigurationsfiler

Vår idé är att konstruera ett enkelt språk där preprocessorsteget är inbyggt i konfigurationsinläsningen. En driftmiljö kan därmed sägas realisera en lokal variant av den globala konfigurationen vid inläsning genom att evaluera uttryck och enklare kontrollstrukturer.

Enkelt uttryckt innebär detta att samma konfigurationsfil används i samtliga målmiljöer men att de konkreta konfigurationsvärden som används är beroende på den miljö filen läses in i.

1.2 Domänspecifika språk

Generalla programmeringsspråk som C, Java och Haskell är byggda för att kunna lösa alla typer av programmeringsuppgifter. Detta genom att ge programmeraren tillgång till alla operationer och den flödeskontroll som krävs för ett turingkomplett språk. Generella programmeringsspråk behöver dock inte vara optimala för alla typer av problem. Program kan bli mer komplicerade eller få en omständig struktur då problemet måste anpassas till programmeringsspråkets ramar och regler.

Ett domänspecifikt språk är ett programmeringsspråk byggt för och begränsat till en specifik problemdomän. Det kan vara svårt att definera vad ett domänspecifikt språk är var gränsen mellan ett domänspecifikt och ett generellt programmeringsspråk bör dras. Följande definition har förslagits av Deursen et al.[DKV00].

> "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

Domänspecifika språks semantik är starkt kopplade till problemen de är byggda för att lösa. De är ofta konstruerade på ett sådant sätt att de skall kunna användas av experter inom den specifika problemdomänen utan att nödvändigtvis kräva annan form av programmeringskompetens.

Domänspecifika språk är ofta små språk med ett begränsat antal notationer och abstraktioner[DKV00]. Domänspecifika språk är väldigt ofta deklarativa språk, eller rent utav specifikationsspråk. Det är också vanligt att domänspecifika språk används för att generera program[BJ79]. Domänspecifika språk kan också användas som ett lager över ett annat språk för att sedan transformeras ner till grundspråket, som till exempel SASS.

Exempel på domänspecifika språk är SQL för databasfrågor; reguljära uttryck för lexikala analysatorer; TEX för typsättning; HTML och CSS för webbsidor och YACC för parsers.

1.2.1 Implementationstekniker för domänspecifika språk

Domänspecifika språk kan delas upp i de två kategorierna externa- och inbäddade språk, också kallade interna språk.

Externa domänspecifika språk

Att implementera ett externt domänspecifikt språk innebär att skapa ett helt nytt fristående språk, med allt vad detta innebär. Alla steg måste göras från grunden inklusive lexikal analys och parsning.

Fördelarna med ett externt språk är att det ger skaparen total frihet gällande syntax och utformning. Detta är den implementationsteknik som ger skaparen störst möjlighet att anpassa sitt språk till den specifika domänen.

Det finns flera möjliga tekniker för att implementera externa språk, vilka vi kommer till i avsnitt 1.3 (s. 4). Implementationen av ett externt språk är dock ofta komplicerat och tidskrävande. Avsaknaden av färdiga utvecklingsverktyg för det nya språket kan också vara ett problem.

Inbäddade domänspecifika språk

Inbäddade domänspecifika språk implementeras i ett redan befintligt programmeringsspråk och kan liknas vid ett API[1] för ett domänspecifikt ändamål.

Benämningen inbäddat språk myntades av Hudak[Hud96], men språkbruket varierar och benämningen *internt domänspecifikt språk* förekommer också.

Denna implementationsteknik ger utvecklaren mycket gratis då inläsning, typsystem med mera redan finns tillgängligt i värdspråket. En annan fördel är möjligheten att använda existerande verktyg skrivna för värdspråket. Priset implementatören betalar för detta stöd är minskade möjligheter att utforma det egna språket efter egna önskemål och den givna domänen.

Ett exempel på ett inbäddat språk är NHibernate som är en ORM-lösning[2] för .NET. Följade kodexmpel i C# är ett exempel på hur en lista med blogginlägg hämtas ur en databas[3].

```
var blogs = s.CreateCriteria<Blog>()
    .Add(Subqueries.PropertyIn("id",
        DetachedCriteria.For<Post>()
            .Add(Restrictions.Eq("Title","NHibernate Rocks"))
            .SetProjection(Projections.Property("Blog.id"))
        ))
    .List<Blog>();
```

[1] Application Programming Interface

[2] Object-Relational Mapping

[3] Exempel hämtat från http://ayende.com/Blog/archive/2009/05/19/nhibernate-queries-examples.aspx

Koden följer helt och hållet syntaxen i C#, men NHibernates funktion är starkt knuten till en specifik domän. Bakom fasaden bygger denna kodsnutt upp en SQL-fråga som sedan körs mot en uppkopplad databas.

1.3 Implementationstekniker för parsning

1.3.1 Parsergeneratorer

En parsergenerator utgår från en beskrivning av syntaxen hos grammatiken som skall implementeras. Denna beskrivning kompileras sedan till källkod i det programmeringsspråk som valts som mål, vilken sedan kompileras och länkas in i målprogrammet.

Lex/Yacc

Lex och Yacc är två tätt sammankopplade domänspecifika språk för att generera lexikala analysatorer respektive parsers[BJ79]. Det finns idag flera implementationer av dessa språk, men GNU-projektets flex och bison är de vi nyttjat.

I Lex beskrivs tokeniseringsregler och matchning med reguljära uttryck och visas enklast med ett kort exempel.

Denna lexikala analysator skriver ut meningar definierade som en följd mellanslagsseparerade strängar av bokstäver avslutade med en punkt. Rader som innehåller andra tecken (vilka matchas med regeln '.') ger ingen utmatning.

```
letter   [a-zA-Z]
word     ({letter})+
sentence ({word}[ ])*({word})[\\.]

%%
^({sentence}) { printf("%s\n", yytext); }
. { }
%%
```

Med Yacc definieras produktioner och vilka regler som gäller för dessa utifrån en syntax som till mångt och mycket liknar BNF[ASU86, s. 257]. Kombinerat ger detta ett system för att generera parsers där grammatiken definieras i Yacc och den lexikala analysen i Flex.

```
%token SENTENCE
%%
sentence: SENTENCE '\n' { }
          |
          ;
%%
```

Exemplet ovan skulle kunna kombineras med en Lex-definition som istället för att endast skriva ut matchande meningar även returnerar produktionen `SENTENCE`.

Lex och Yacc har inspirerat en uppsjö verktyg även för andra plattformar, däribland JFlex, CUP och ANTlr för Java.

1.3.2 Parserkombinatorer

Det som kännertecknar parserkombinatorer är att de utgörs av högre ordningens funktioner med infixnotation[Wad85], beskrivna av Philip Wadler redan 1985. Då dessa baseras på egenskaper primärt funna i funktionella språk är det inte förvånande att det är i den världen vi först finner dem.

En parserkombinator tar en eller flera funktioner som i sin tur accepterar en sträng som indata och returnerar en två-tupel som innehåller en lista av inlästa symboler samt den del av indatasträngen som funktionen inte konsumerat.

Det verktyg vi valt att använda är Parsec, ett monadiskt parserkombinatorbibliotek för Haskell[ULM01] som utnyttjar Haskells lata evaluering av uttryck. Till skillnad från exempelvis Lex/Yacc så skrivs all kod i ren Haskell utan något mellanliggande kompileringssteg. Parsec har inspirerat ett flertal andra parserkombinatorbibliotek för bland annat Java, F# och Erlang. Parsec är öppen källkod licensierad under BSD-licensen.

Kapitel 2

Metod

2.1 Kontextfria grammatiker

Det finns olika klasser av grammatiker, men det vi ägnade oss åt var att konstruera var en kontextfri sådan. En kontextfri grammatik är en grammatik som består av följande beståndsdelar.

- Icketerminaler - Symboler som definierats i termer av terminaler, icketerminaler samt strängar av dessa.
- Terminaler - Tecken i alfabetet som grammatiken verkar över.
- Strängar av icketerminaler - Listor av icketerminaler.
- Strängar av terminaler - Listor av terminaler.

I en kontextfri grammatik kan varje icketerminal uttryckas som en följd terminaler, icketerminaler samt strängar av dessa. Dessutom betraktas en specifik icketerminal som startsymbol utifrån vilken syntaxträdet byggs upp[SSS88].

Reglerna för en kontextfri grammatik skrivs ofta på Backus-Nauer-form (BNF). För exempel på hur detta kan se ut se avsnitt 2.3.1 (s. 9).

2.2 Haskell

Haskell är ett statiskt typat, rent funktionellt programmeringsspråk med lat evaluering.

2.2.1 Monader

En monad är ursprungligen en algebraisk struktur som används inom den abstrakta algebran. I funktionell programmering är det en typ av abstrakt datastruktur som används för att sammalänka operationer[BO08, s. 186]. Detta för att få operationerna att exekveras i ordning, trots att språket använder sig av lat evaluering.

Det var från början ett sätt att få in in- och utmatningsoperationer i Haskell. Dels för att kunna utföra inmatnings- och utmatningsoperationer som inte returnerade något resultat överhuvudtaget och dessutom i sekventiell ordning, men också för att separera kod med sidoeffekter från övrig kod[BO08, s. 188]. Monader är har även visat sig vara praktiska i andra sammanhang, som exempelvis vid parsning.

I Haskell är en monad en datatyp av typklassen Monad[Jon03]. En monad är egentligen bara en inkapsling runt en datatyp som implementerar två väldigt rudimentära operationer, nämligen sekvensoperatorn "`>>=`" (benämns *bind* eller *chain*) och `return`. Definitionen i Prelude[1] är som följer[Jon03].

```
class Monad m where
  -- bind/chain
  (>>=)   :: m a -> (a -> m b) -> m b
  -- sequence
  (>>)    :: m a -> m b -> m b
  -- inject
  return  :: a -> m a
  -- abort
  fail    :: String -> m a

  m >> k  =  m >>= \_ -> k
  fail s  = error s
```

Bindoperatorn matar hela tiden vidare resultatet av monaden i vänsterledet till monaden i högerledet. Detta gör att monaden till höger om operatorn beror på monaden till vänster, vilket leder till att exekveringsordningen blir från vänster till höger. När flera operationer sätts ihop i en kedja, skulle det kunna liknas vid att en tråd träds igenom hela kedjan från slutet till början där alla operationer beror på resultatet av den föregående. På så sätt blir alla operationer utförda i rätt ordning när hela monaden sedan evalueras.

Funktionen `return` kapslar in ett värde i en monad och används vanligvis i slutet av en kedja av monader. Namnet `return` är inte så bra eftersom monaden inte på något sätt avbryts med operationen. Vad `return` gör är att den injicerar en värdetyp in i en monadtyp. Ett lämpligare namn hade därför varit *inject*[BO08, s. 329].

Det finns en monad inbyggd i Haskell med namnet `Maybe`. Den är definierad på följande sätt[Jon03].

```
data Maybe a = Just a | Nothing

instance  Monad Maybe  where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
```

[1] Haskells standardbibliotek

```
return          =  Just
fail s          =  Nothing
```

Denna monad kan användas för att representera en misslyckad (`Nothing`) eller en lyckad (`Just a`) operation. Maybe-monadens "`>>=`" operator är implementerad på ett sådant sätt att om den vänstra monaden är `Nothing`, returneras `Nothing` och det som står till höger om operatorn kommer inte att evalueras.

Det är på denna Maybe-monad som Parsec är uppbyggt. Om en parser accepterar symbolsträngen med indata returneras `Just a`, där a är det genererade resultatet, annars returneras `Nothing`.

2.3 Parsec

Parsec är ett monadiskt parserkombinatorbibliotek för Haskell som släpptes som öppen källkod 1999. Med Parsec konstrueras parsers genom att med hjälp av kombinatorer sätta samman existerande parsers som då agerar som byggstenar, en modell som till mångt och mycket efterliknar konstruktionen av kontextfria grammatiker.

Med Parsec konstrueras LL(1)-parsers[Lei01]. Analysen sker uppifrån och ned, med vilket menas att från en rotnod expanderas rekursiva parsers och ett parseträd konstrueras[ASU86, s. 41]. Parsec stöder backtracking, med vilket menas att parsern kan backa och behandla en sträng flera gånger, men det måste anges explicit när det skall användas.

En parser i Parsec är en monadisk funktion vars monad ärver från en basklass definierad i Parsec. En parser arbetar hela tiden mot indata i form av en lista med symboler. I vår implementation arbetar vi direkt mot en lista av teckentypen `char`, men Parsec är byggt så att egenhändigt konstruerade symboltyper kan användas.

En parser är en sammansättning av andra parsers, som i sin tur består av andra parsers, och så vidare ner till den lägsta nivån där parsern kontrollerar om en enskild symbol satisfierar ett villkor eller inte.

En parser konsumerar symboler från indatan så länge de satisfierar de givna villkoren ställda av parsern. Om en symbol inte uppfyller kraven misslyckas parsern. De symboler som konsumerats av parsern fram tills misslyckandet kommer inte att återställas, förutsatt att backtracking inte explicit används.

Parsers kombineras med den binära operatorn "`<|>`". Den fungerar så att om parsern till vänster om operatorn misslyckas kommer parsern till höger att köras. Det går att sätta ihop obegränsat antal parserns med hjälp av denna kombinator.

2.3.1 Exempel

Relationen mellan den grammatik som implementeras och resulterande Haskell-kod visas enklast med några exempel.

Vi presenterar här ett exempel för hur en parser som accepterar meningar implementeras med Parsec. En mening är i det här fallet ett eller flera ord, separerade med mellanslag som avslutas med en punkt.

Denna grammatik definierar icketerminalen "`word`" som en följd av bokstäver. I BNF skriver vi detta på följande sätt.

```
word ::= [a-zA-Z]+
```

Parsern "`word`" nedan implementerar grammatiken på ett rättframt sätt med hjälp av två parsers som följer med Parsec: "`many1`" är en högre ordningens parser som matchar en eller flera av den parser som skickas in som argument, i detta fall "`letter`" som i sin tur matchar vanliga bokstäver.

```
word::Parser String
word = many1 letter
```

En naturlig utökning av ovanstående grammatik är att sätta samman ord till meningar där mellanslag separerar ord och punkt avslutar en mening. Grammatiken skrivs i BNF på följande sätt.

```
sentence ::= word ' ' sentence | word '.'
```

Motsvarande Haskell- och Parseclösning kan se ut på följande sätt:

```
word'::Parser String
word' = char ' ' >> word -- Mellanslag följt av ett ord
                         -- >> är sekvensoperatorn i Haskell

sentence::Parser [String]
sentence = do {
      h <- word       -- Meningen påbörjas med ett ord,
    ; s <- many word' -- fortsätter med noll eller flera ord som
                      -- påbörjas med mellanslag
    ; char '.'        -- och avslutas slutligen med punkt.
    ; return (h:s)    -- Meningens alla ord returneras som en lista
    }
```

Som synes följer uttryckssättet med Parsec nära det i motsvarande BNF-definition. Att köra parsern "`sentence`" på strängen "Powered by Parsec." resulterar i listan ["Powered", "by", "Parsec"].

Ytterligare en uttökning som även accepterar heltal som ord, med följande förändring till grammatiken.

```
word ::= [a-zA-Z]+ | [0-9]+
```

Att från detta modifiera sentence-parsern kräver endast följande förändring, där vi introducerar kombinatorn "`<|>`" vilken i första hand kör parsern till vänster, men om denna misslyckas istället försöker med parsern till höger. Misslyckande är i detta fall definierat såsom att en parser inte konsumerar någon indata.

```
word::Parser String
word = many1 letter <|> many1 digit
```

Parsers kan göras väldigt kompakta i Haskell och med Parsec. För att ge ett exempel på detta ger vi här därför en alternativ implementation av ovanstående exempel. Denna implementation använder fler hjälpfunktioner från Parsec och dessutom *applicatives*, en variant av monader i Haskell. Dessa exempel är bara till för att ge prov på hur uttrycksfull kombinationen Haskell/Parsec är och en mer ingående förklaring ges därför ej.

```
-- Accepterar meningar med ord bestående av enbart bokstäver
sentence1 = (many1 letter) `sepBy1` space <* char '.'

-- Accepterar meningar med ord bestående av enbart
-- bokstäver eller enbart siffror
sentence2 = (many1 letter <|> many1 digit) `sepBy1` space <* char '.'
```

Kapitel 3

Design och implementation

3.1 Språkdefinition

Målsättningen i utformandet av språket var att söka en så hög uttrycksfullhet med så låg implementationskomplexitet som möjligt för att möjliggöra implementation inom de givna tidsramarna.

För att inte begränsa namnrymden för nycklar används inga nyckelord, utan språkets hela syntax består av specialtecken.

Parsec presterar bäst då den tolkade grammatiken är prediktiv och LL(1)[Lei01], något vi utgick ifrån vid utformningen.

3.1.1 Exempel

Nedan följer ett exempel på en konfigurationsfil med tillhörande resultat efter evaluering givet en uppsättning inparametrar. De fördefinierade parametrarna anges i exemplet med initiala versaler.

Namn	**Datatyp**	**Värde**
Profile	String	“test”
TestWithotutMT	Bool	False
FastDebug	Bool	False
Cores	Integer	4
ThreadsPerCore	Integer	6

Konfigurationsfilens innehåll följer nedan.

```
# Maximalt antal trådar som programmet kan starta
maxThreads ? $Profile == "test" && $TestWithoutMT -> 1,
             $Profile == "debug"                  -> 1,
             _                -> $Cores * $ThreadsPerCore;

# All konfiguration för loggning grupperas tillsammans
logging {
  # Filnamnet för loggfilen
  output = "log_" + $Profile + ".txt";
  # Hur mycket skall skrivas till loggen?
  logLevel ?
    $Profile == "debug" && !$FastDebug        -> "warnings,errors,debug",
    $Profile == "debug" || $Profile == "test" -> "warnings,errors",
    _                                         -> "errors";
};
```

Detta är konfigurationen som erhålls efter evaluering av konfigurationsfilen.

Namn	**Datatyp**	**Värde**
maxThreads	Integer	24
logging.output	String	”log_test.txt”
logging.logLevel	String	”warnings,errors”

3.1.2 Grammatik

För att börja beskrivningen av vår grammatik måste först icketerminalen “`ws`” beskrivas.

Språket stödjer radkommentarer överallt där blanktecken är tillåtet. Kommentarer börjar vid brädgårdstecken (‘#’) och fortsätter sedan tills nästa rad eller filslut. Icketerminalen “`ws`” (förkortning av whitespace, det vill säga blanktecken) används för att beskriva platser där blanktecken och kommentarer är tillåtna men inte nödvändiga.

```
ws           ::= [\w]* comment ws | [\w]*
comment      ::= "#" (^\n)* "\n"
```

Vår språkdefition börjar i rotnoden “`config`”. Konfigurationsfilen kan börja med flera blanktecken och därefter icketerminalen “`statements`”, vilket är en pluraliserad form av icke-terminalen “`statement`”. Denna icketerminal är den enda som kan existera på rotnivå. Ett “`statement`” består av en *identifierare* följt av en *deklaration*, som ges av icketerminalen “`decl`”. Alla icketerminaler av typen “`statement`” avslutas med ett semikolon.

```
config       ::= ws statements
statement    ::= identifier ws decl ";"
statements   ::= statement ws stmnts | empty
```

En identifierare i vår grammatik definieras enligt nedanstående reguljära uttryck.

```
identifier     ::= [_a-zA-Z][a-zA-Z0-9_]*
```

Deklaration efter identifieraren kan vara en av tre typer: värdes-, fall- eller blockdeklaration.

- **Värdesdeklaration** - Definierar ett värde på variabel med namnet givet av identifieraren.
- **Falldeklaration** - Evaluerar en lista av *fall* och värden för varje respektive fall. Det första fallet som evalueras till *sant* ger det värde som slutligen sätts på variabeln med namnet som getts av identifieraren. Det måste alltid finnas ett standardvärde angivet i slutet av listan som väljs om inget annat fall uppfylls.
- **Blockdeklaration** - Skapar ett nytt block för att logiskt gruppera variabler under ett gemensamt prefix. Dessa kan definieras i flera nivåer.

Dessa deklarationer definieras i BNF på följande sätt.

```
decl         ::= valuedecl | casedecl | blockdecl
valuedecl    ::= ":" ws expr
casedecl     ::= "?" ws cases "_" ws "->" ws expr
blockdecl    ::= "{" ws statements "}"
```

Icketerminalen "`cases`" definierar en lista med fall för en falldeklaration.

```
cases        ::= expr ws "->" ws expr ws "," ws cases | empty
```

Grammatiken stödjer uttyck med aritmetik och booleska operationer mellan olika datatyper. De datatyper vi stödjer är strängar, heltal, flyttal och booleska värden. Det går även att referera till deklarerade variabler och använda dem i uttyck eller direkt vid värdesdeklarationer. Referenser har formen "`$foo.bar`". Alla referenser börjar med tecknet '$', sedan en sammanslagen lista för varje nivå av block varibeln är deklarerad i och sist kommer variabelnamnet.

```
factor       ::= StringLiteral | Integer | Float | boolean | propref
boolean      ::= "True" | "False"
propref      ::= "$" (propname ws "." ws)* propname
```

Vi stödjer de vanliga operationerna med både unära och binära operatorer med syntax och prevalensordning lånad från språket C. Parenteser kan användas för att ändra evalueringsordningen i uttryck.

```
expr         ::= "(" ws expr ws ")" | factor | unaryop ws expr |
                 factor ws binaryop ws factor
unaryop      :: = "-" | "!"
binaryop     ::= arithmeticop | testop
arithmeticop ::= "*" | "+" | "-" | "/"
testop       ::= "&&" | "||" | "==" | "!=" | "<" | ">" | "<=" | ">="
```

3.1.3 Semantik

Språket definieras inte endast utifrån de syntaktiska reglerna ovan, utan även av en mindre uppsättning semantiska regler.

- Nycklar kan endast deklareras en gång.
- Nycklar kan endast slås upp om de deklarerats längre upp i filen.
- Aritmetiska operationer mellan heltal returnerar heltal.
- Aritmetiska operationer mellan flyttal returnerar flyttal.
- Aritmetiska operationer som blandar flyttal och heltal returnerar flyttal.
- Det går ej att göra likhetsjämförelser på flyttal.
- Om inget fall matchas i en falldeklaration returneras standardvärdet ('_').

3.2 Implementation

Vi byggde en tolk för ett statiskt typat konfigurationsspråk, antingen att använda som en inlänkad modul i ett större program eller ett fristående program som förbehandlar en konfigurationsfil för senare inläsning.

Parserns tillstånd fångas av en monad, en arkitektur kraftigt influerad av Hudak [Hud96].

Implementationen är uppdelad i följande moduler.

- **ConfParser** - Innehåller funktioner för lexikal analys och parsing.
- **ConfParserAS** - AS står för *Abstract Syntax*. Innehåller definitioner av datatyper och modeller. Implementerar även aritmetiska och logiska operationer på de inbyggda datatyperna.
- **ConfParserMain** - Kommandoradsverktyg för att läsa in konfigurationsfiler och visa alla parametrars evaluerade värden.

Kapitel 4

Resultat

Språket vi byggde utgör ytterligare ett sätt att angripa problemet med konfigurationsdistribution genom att införa en rikare syntax jämfört med rena datalagringsspråk, detta i form av aritmetiska uttryck, falldeklarationer och typsystem.

Källkoden till tolken finns att ladda ned på `http://www.nada.kth.se/~hmatt/dkand10/intrprt.tar.gz`. Koden är utvecklad och testad med GHC 6.10.2 och Parsec 2.1.0.1.

4.1 Diskussion

4.1.1 Parsec

Vi lät Parsecs styrkor och svagheter influera grammatiken hos vårt språk och stötte därför inte på några oöverbryggliga svårigheter vid implementationsarbetet. Vi utnyttjade utöver detta Parsecs stöd för att implementera infixoperatorer, vilket underlättade arbetet med att stödja aritmetiska uttryck. Namngivningsoperatorn "`<?>`" som används för att bestämma syntaxfelmeddelanden både underlättade felsökningsarbetet och lät oss på ett enkelt sätt införa stöd för meningsfulla felmeddelanden i vår tolk.

4.1.2 Haskell som implementationsspråk

Under implementationsarbetet fann vi att några egenskaper hos Haskell underlättade arbetet avsevärt. Dessa egenskaper är inte nödvändigtvis direkt kopplade till Parsec men ändå väl värda att ta upp.

Mönstermatchning

Mönstermatchning innebär att en polymorf funktion kan innehålla specialfall som motsvarar anrop med en viss signatur. Detta gjorde kontrollen av de semantiska reglerna, som i vårt fall till stor utsträckning behandlar vilka typer och operationer som går att blanda, enkel att både skriva och läsa.

Ogiltiga uttryck hanterades av det allmänna uttrycket samtidigt som giltiga uttryck evaluerades baserat på de ingående typerna i specialfallen. Detta gav oss möjlighet att enkelt skilja giltiga uttryck från ogiltiga genom att betrakta typerna hos argumenten till evalueringsfunktionerna.

Algebraiska datatyper

Algebraiska typer, i vårt fall framförallt i form av behållare för primitiva datatyper (t.ex. heltal eller flyttal), användes i stor utsträckning för att kunna mönstermatcha de resulterande uttrycken. Detta gjorde det enkelt att resonera kring de semantiska reglerna samt gradvis införa stöd för dessa.

4.1.3 Fördelar med vårt angreppssätt

Språkets syntax är förhållandevis kompakt då nycklar och värden inte dupliceras (jämför med t.ex. XML). Kravet på standardvärden i falldeklarationer får till effekt att om en nyckel definierats så kommer den finnas tillgänglig i alla miljöer.

Aritmetiska uttryck och val kan användas för att ge den som konfigurerar större inflytande över applikationens beteende.

4.1.4 Nackdelar med vårt angreppssätt

Avsaknad av nyckelord gör syntaxen svårläst. Påtvingade standardvärden gör att det inte möjligt att stödja en situation där konfigurationsvärden som endast har mening i en delmängd av driftmiljöerna kan utelämnas helt i övriga.

4.2 Slutsatser

Haskell i allmänhet och kombinerat med Parsec i synnerhet ger användaren verktyg att implementera externa domänspecifika språk med rik syntax för att lösa problem i en given domän, något vi demonstrerat genom ett exempel.

Litteraturförteckning

[ant] *Apache Ant Manual, hämtad 2010-02-03.* http://ant.apache.org/manual/CoreTasks/property.html.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition, 1986.

[BJ79] Stephen Johnson Bell and Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[BO08] John Goerzen Bryan O'Sullivan, Don Stewart. *Real World Haskell.* O'Reilly Media, 1 edition, 2008.

[DKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN NOTICES*, 35:26–36, 2000.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, Cambridge, England, 2003.

[Lei01] Daan Leijen. Parsec, a fast combinator parser. 2001.

[res] *Caucho Resin Manual, hämtad 2010-02-03.* http://www.caucho.com/resin-3.1/doc/resin-tags.xtp.

[SSS88] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing theory. Vol. 1: languages and parsing.* Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[ULM01] Daan Leijen University, Daan Leijen, and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, 2001.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

Bilaga A

Ordlista

BNF Backus-Naur-Form, ett språk för att beskriva kontextfria grammatiker

DSL Domänspecifikt språk, från engelskans Domain Specific Language.

LALR-parser En variant på LR-parser som ofta genereras av en parsergenerator. LALR(k) syftar till en LALR-parser som tittar k tecken framåt.

LL-parser Uppifrånochnedparser som läser från vänster till höger och bygger syntaxträd från vänster. LL(k) syftar till en LL-parser som tittar k tecken framåt.

LR-parser Uppifrånochnedparser som läser från vänster till höger och bygger syntaxträd från höger. LR(k) syftar till en LR-parser som tittar k tecken framåt.

Rekursiv medåkning En implementationsmetod för uppifrånochnedparsers baserad på rekursion där det resulterande programmet till mångt och mycket liknar den grammatik som implementeras.

Uppifrånochnedparser En parsertyp för grammatiker som kan läsas som hierarkiska trädstrukturer.