Fakulteten för ekonomi, kommunikation och IT

Programspråk

Laboration 4

Datum: 2007-09-03
Namn: Henrik Bäck
Mathias Andersson
Kurs: DAV C02

Innehållsförteckning

Sammanfattning

Den här rapporten tar upp för och nackdelar samt jämför de tre spårken C++, Prolog och Lisp.

De tre språken jämförs genom en implementation av en Parser och en Lexer för Pascal. Mängden programkod mäts genom att räkna antalet programkodrader. Det visar sig att C++ har flest antalet

Konceptet med att ha någon form av databas över alla nyckelord återkommer i samtliga språk trots att den implementeras på olika sätt. Valet hur tokens skall hanteras ligger främst hos programmeraren och framförallt C++ där densamma har större valfrihet. I C++ skulle en bättre lösning varit att ha programmerat implementation av tokes med hjälp av strukturer eller eventuellt objekt. Om strukturer valts hade implementatioen i C++ blivit än likare den i Lisp.

En annan skillnad mellan de tre implementationerna är hur Parsern begär tokens från Lexern. I C++ finns en funktion som Parsern använder sig av för att få tokens. Denna funktion returnerar nästa token i strömmen. På så sätt kan Parsern bara begära en ny token varje gång den vill gå framåt i kontrollen. I Lisp fungerar det på ett likadant sätt. Det är sedan Prolog som har en helt annan implementation på denna punkt. I Prolog kommer en lista med tokens att generas från början och denna kommer sedan, i sin helhet, att skickas in till Parsern. Varje funktion i Parsern tar sin del av listan och skickar vidare resten.

C++ har den fördelen att det är ett Imperativt språk. Eftersom de flesta idag lär sig imperativa språk från grunden blir det lättast att förstår sig på detta och hur denna implementation fungerar. En annan fördel med C++ är att omfattningen av språket. Många använder i dag C++ vilket gör det till ett kraftigt språk med mycket att hämta. C++ har också fördelen av att det går att använda för att göra i stort sett allt inom datorteknik. På grund av C++s hårdvarunära implementation så kan koden göras väldigt resurseffektiv. Prolog har den stora fördelen att det är deklarativt. På så sätt slipper programmeraren att tala om exakt hur operationerna skall utföras, utan det räcker med att ange vad som skall göras. Detta ger att programmeraren kan koncentrera sig på vad som skall utföras istället för att ägna tid åt hur det skall utföras. Lisps stora fördel är att det klart och tydligt klarar av att överföra planering och idéer om hur programkod skall fungera direkt till hur det faktiskt skrivs. Tyvärr verkar Lisps fördelar sluta här igämförelse med de andra språken. Dock skall denna fördel inte underskattas då det är en stor konst att skriva bra datorprogram.

Inledning

Den här rapporten kommer att visa närmare på likheter och skillnader mellan programspråk i olika kategorier. För samtliga språk har en parser för förenkald Pascal-kod implementerats. Dock skiljer sig implementationen och implementationens funktionalitet mellan de olika språken. Av denna anledning kommer analysen att begränsas till de delar som är lika implementerade i samtliga tre språk.

De språk som kommer att tas upp i den här rapporten är C++, Prolog samt Lisp. De punkter som rapporten kommer att titta närmare på är hur språket används för att implementera en viss funktionalitet, hur mycket programkod som behöver skrivas för att uppnå denna funktionalitet samt för och nackdelar med språken.

Vad kommer att jämföras?

Den här rapporten kommer jämföra hur de olika språken används för att åstakomma en och samma funktionalitet. I det här fallet har en Parser och en Lexer implementerats i de olika språken.

Vidare kommer även aspekter på tid och mängden programkod att jämföras. Några slutsatser från detta kommer också att dras.

Återkommande definitioner

Den här rapporten kommer återkommande gånger använda ett antal begrepp. För att bringa klarhet i dessa begrepp kommer de att definieras i denna sektion.

Programkodsrad (NCLOC)

På grund av att de tre språk som tas upp i denna rapport har olika sätt att avsluta ett programpåstående så kommer här en utförlig beskrivning på vad som kommer att räknas som en NCLOC (Non-commented Lines Of Code) för respektive språk. Att räkna antalet rader kod som ett en viss funktionalitet implementerar i ett visst språk kan användas för att se hur mycket fysiskt programmeringsarbete som behövs för att färdigställa programmet. Detta behöver dock inte visa på att ett språk med fler rader kod än ett annat är svårare att implementera.

C++

För C++ kommer en (1) programkodsrad att räknas som en (1) rad vilken innehåller ett programpåstående och som har avslutats med ett semikolon. Till detta räknas även slingor, loopar samt val. Dock räknas inte inkluderingar av bibliotek eller definitioner, till denna defintion.

Prolog

För Prolog kommer ett (1) predikat att räknas som en (1) programkodsrad. Detta innebär att även ifall programkoden är skriven över flera fysiska rader för ett och samma predikat kommer dessa enbart att räknas som en (1) rad.

Lisp

För Lisp kommer en (1) programkodsrad att räknas som det som kan skrivas med hjälp av en funktion och är definierad i en egen funktionskropp (`defun`). Till detta räknas även själva definitionen av funktionen. Nedan finns ett exempel för att förtydliga det hela.

```
(defun test()
            (fn1 'fn)
            (fn2 t))
```

Exemplet ovan visar på två (3) programkodsrader för en Lsip-funktion.

Språkets effektivitet

Rapporten kommer att använda sig av begreppet effektivitet i samband med ett programspråk. I det här sammanhanget är det viktigt att förklara exakt vad som menas med detta.

Tolkningen i den här rapporten är sådan att man enbart kan använda begreppet effektivare i jämförelse mellan två språk. Ett språk är då effektivare än ett annat om man kan utföra exakt samma funktionalitet i båda språken medan antalet NCLOC är färre i det effektivare språket än det andra. Då sägs det att ett språk är effektivare än ett annat.

Antalet funktioner

Rapporten kommer att använda sig av antal funktioner som är definierade som ett mått. För att bringa klarhet i detta begrepp så kommer här en förklaring på vad respektive språk definierar som en funktion.

C++

Varje C++-funktion kommer att räknas som en funktion.

Prolog

Varje predikat i Prolog räknas som en funktion.

Lisp

Varje Lisp-funktion definierad med `defun` kommer att räknas som en funktion.

Tid för implementation

Den tid det tagit att implementera samma funktionalitet i samtliga språk kan vara av visst värde att räkna på då det kan visa på hur effektiviteten hos ett språk är. Dock måste man i detta fall ha i åtanke att både Lisp och Prolog har varit okända språk för programmerarna vid den inledande fasen av Parserns implementation. Detta innebär att tid har gått åt till att lära sig språket istället för att utföra själva implementationen. Dock kommer tid att vara en aspekt i denna rapport och den kommer att beräknas i hela timmar och vara estimerad så gott det går till att vara den tid det har tagit att programmera funktionaliteten.

Tillförd programkod

Speciellt för implementationen i Lisp är att den bygger på tidigare kod. Det som har skett är att denna kod har kompletterats för att uppfylla samtliga krav för Parsern. Det här ger att det inte går att beräkna hur lång tid det skulle tagit att implementera de

delarna i Lisp som redan var klara. Därför kommer motsvarande delar att räknas bort från C++ och Prolog när jämförelse av tiden sker. Om inte detta görs skulle en felaktig bild av tiden det tar att skriva något i respektive språk att vara missvisande.

Jämförelse av språken

Rapporten kommer att belysa de delar som är implementerade i samtliga tre språk. Det som kommer granskas närmre är hur Parsern är implementerad samt den del av Lexern som läser in en programfil och generar tokens till Parsern.

Representation av tokens

Eftersom alla tre språken fungerar på helt olika sätt måste även representationen av Pascals nyckelord ske på olika sätt. Dock har de gemensamt att nyckelorden kommer att representeras av tokens i någon form. I den här delen av programmet skiljer sig implementationerna som mest.

C++

I C++ kommer alla token att representeras av ett tal. I detta fall måste ett system utarbetas som fungerar både för enskilda tecken och för de nyckelord som finns i språket. Eftersom alla tecken kommer att kunna motsvara ett tal mellan 0 och 255 (med antagandet om att vi följer ASCII) så kan alla tal över 255 användas för att adressera de nyckelord som behöver definieras. På detta sätt blir det möjligt för tokens att representera nyckelord bestående av antingen enskilda tecken eller hela ord. I de fall då ett ord används har dessa lagrats i en databas och matchas mot ett tal över 255.

För att lyckas med detta behövs en inläsningsalgoritom, som utifrån bestämda regler, kan bestämma ifall den text som finns i form av strängar är nyckelord eller om den enbart skall läsa ett tecken. I C++ utgörs denna av en sträng vilken innehåller hela det program som skall genomsökas. Lexern läser in en fil till strängen och använder sedan en algoritm för att bestämma, vid varje förfrågan om token från parsern, vad den skall returnera.

Samtliga möjliga nyckelord finns lagrade i en tabell. Med hjälp av denna tabell matchas sedan nästa teckenkombination i strängen för att avgöra om det är en token eller ej. Om så är fallet returneras denna token i annat fall så kommer enbart det aktuella tecknet att returneras.

Prolog

I Prolog representeras alla nyckeord av dess riktiga namn i form av en ASCII-kombination. I Prolog kan detta enkelt göras genom att skapa en databas över alla giltiga nyckelord.

Inläsningen av programmet sker till en lista. Listan kommer att motsvara allt innehåll i programfilen. Över den här listan körs sedan ett predikat som matchar listans innehåll mot databasen över giltiga nyckelord. På detta sätt listans innehåll översättas till tokens. Om ett ord i listan inte matchar något nyckelord tolkas det som en id-token.

Till Parsern skickas sedan hela listan med tokens. Parsern plockar sedan själv den första positionen i listan.

Lisp

I Lisp, likt Prolog, representeras varje token av sitt respektive namn. Det finns en funktion som har möjlighet att matcha en sträng mot ett nyckelord. Den här funktionen returnerar sedan denna token. Finns ingen match kommer funktionen att returnera en ”unknwon”-token.

Inläsningen av programmet sker efter hand. Varje gånge Parsern begär en ny token från Lexern så kommer denna att läsa nästa tecknenkombination från filströmmen, matcha denna inläsning mot listan av tokens och returnera korrekt token.

Slutsats

Här ser man direkt att det finns skillnader i språkens uppbyggnad men också skillnad i hur man bör tänka i de olika språken. Konceptet med att ha någon form av databas över alla nyckelord återkommer i samtliga språk trots att den implementeras på olika sätt.

Valet hur tokens skall hanteras ligger främst hos programmeraren och framförallt C++ där densamma har större valfrihet. Vissa val som gjorts angående lagring av tokens kunde ha varit mer genomtänkta och skulle då ha givit en mer liknande implementation mellan de olika språken. I C++ skulle en bättre lösning varit att ha programmerat implementation av tokes med hjälp av strukturer eller eventuellt objekt. Om strukturer valts hade implementatioen i C++ blivit än likare den i Lisp.

Parsingmetod

I samtliga tre programspråk är Parsern uppbyggd med hjälp av högerrekursion. Detta innebär att implementationerna språken emellan blir mycket lika. I samtliga tre versioner går Parsningen av pascalkoden till ett likartat vis.

I storta drag fungerar det som sådant att en funktion ansorppas som startar parsningen. Efter detta är strukturen ganska rakt fram. Först anropas kontrollen av programhuvudet, här kontrolleras det så att den pascalkod som skall kontrolleras har det förväntade utseendet hos ett programhuvdet. Efter detta kontrolleras variabeldelen och sedan själva programdelen. Dessa kontroller går till på så sätt att nyckelorden som förväntas matchas mot de som finns i filen. Parsern begär en token från Lexern, se tidigare avsnitt, och kontrollerar sedan ifall denna stämmer överens med den som borde stå på denna position. Om så inte är fallet kommer ett felmeddelande skrivas ut och vad som händer efter detta är lite olika mellan de tre implementationerna.

I C++ fortsätter parsern att kontrollera resten av programmet på samma sätt medan i Prolog och Pascal avbryts kontrollen för den aktuella filen.

En annan skillnad mellan de tre implementationerna är hur Parsern begär tokens från Lexern. I C++ finns en funktion som Parsern använder sig av för att få tokens. Denna funktion returnerar nästa token i strömmen. På så sätt kan Parsern bara begära en ny token varje gång den vill gå framåt i kontrollen. I Lisp fungerar det på ett likadant sätt. Det är sedan Prolog som har en helt annan implementation på denna punkt. I Prolog kommer en lista med tokens att generas från början och denna kommer sedan, i sin helhet, att skickas in till Parsern. Varje funktion i Parsern tar sin del av listan och skickar vidare resten.

Tid för implementation

För att titta på de delar som är implementerade i samtliga programspråk så är det svårt att säga i vilket språk det har gått snabbast att implementera en Parser. Vad det gäller lexern är det mycket svårt att säga eftersom mestadels av koden för lexern i Lisp var redan klar från start.

I C++ implementerades även en symbolltabell, operationstabell samt en typ-kontroll. Detta har inte gjorts i något annat språk och tiden för detta kan därför inte heller jämföras

För att implementera parsern för de övriga två språken, Prolog och Lisp, åtgick en snartlik mängd tid. När det gäller Lisp fanns också nackdelen med att språket var så pass nytt för programmerarna att det var svårt att hitta de funktioner som är inbyggda i språket. Detta har tagit lång tid i jämförelse med C++.

Tidsdifferensen

Om man bortser från den tid som har lagts på att skriva om språkreglerna för Pascal så att de blir högerrekursiva så återstår enbart den del där dessa regler skall skrivas i respektive språk.

Eftersom samtliga tre språk klarar av rekursion är implementation av detta mycket enkelt. Det som behövs är att skriva om reglerna i programspråket. Här skall det också nämnas att det har tagit längre tid att implementera det hela i C++ eftersom det var det första språket som parsern implementerades i. Det gör att tid har gått till att lösa mindre problem med den högerrekursion som först skapats. När de övriga språken implementerades har det gått mindre tid till detta eftersom programkoden i C++ har kunnat legat till grund för hjälp.

Antalet programkodsrader

För de den delen av Parsern som är implementerad i samtliga språk finns en stor skillnad på mängden programkodsrader som har behövts skrivas.

I C++ har det åtgått 403 antal programkodsrader för att implementera parserns grundläggande delar. Samma delar tar i Prolog 99 programkodsrader och 208 rader i Lisp. Här kan man direkt se att Prolog har en väsentlig mindre mängd programkodsrader jämfört med både C++ och Lisp. Det här säger inte att Prolog skulle vara ett bättre språk än de andra två men det säger helt klart att språket måste vara mer effektivt än de andra.

Att prolog enbart kräver 99 rader för att utföra samma funktionalitet som C++ utför på 403 beror bland annat på att Prolog är deklarativt. Här behövs inga algoritmer skrivas för hur operationerna skall utföras, i alla fall inte i samma bemärkelse som i C++ och Lisp.

Slutsats

Av detta kan det vara svårt att dra en vettig slutsats. Det som trivialt syns är att Prolog har färre antal programkodsrader än de andra två språken. Dock är det så att det tog

programmerarna lika lång tid, se tidigare resonemang, att skriva koden i de olika språket.

Detta måste betyda att färre antal rader inte leder till en mindre tidsåtgång vid själva implemetationstillfället för Prolog-programmet. En anledning till detta kan vara att det för varje programkodsrad krävs det mycket mer tankearbete eftersom varje programkodsrad innehåller mycket mycket mer funktionalitet än i de övriga språken. En annan anledning till de färre antalet raderna kan ha att göra med att Prolog har ett bättre stöd för att matcha strängar och hantera listor än C++.

Lisp kommer mitt emellan C++ och Prolog vad det gäller antalet programkodsrader. Det skulle också kunna sägas att Lsip ligger mitt emellan C++ och Prolog vad det gäller sättet att programmera på. Detta innebär att Lisp har fördelarna med att ha lösare typning än C++ och dessutom har den ett enklare sätt, sett ur ett Imperativt öga, att implementera funktionaliteten på.

Antalet funktioner

Förvånansvärt nog inhåller både C++ och Lisp nästan lika många funktioner. I C++ finns 22 st och i Lisp finns 24. I avstickaren Prolog finner vi hela 99 funktioner.

Eftersom Prologs predikat tolkas som funktioner så är antalet funktioner i denna implementation inte alls förvånande. Att C++ och Lisp däremot har näraå samma antal funktioner är intressant, dock inte särskilt förvånande. Eftersom implememtationerna i C++ och Lisp ligger nära varandra är det högt sannorlikt att de skulle ha ett liknande antal funktioner. Det skulle dock vara möjligt, i C++, att skriva samma funktionalitet med ett mycket mindre antal funktioner. Detta med hjälp av iterationer istället för rekursion. Visserligen skulle samma sak säkerligen gå göra i Lisp men iterationer är inte något för ett funktionellt språk.

Skillnader och likheter mellan språken

Den här delen av rapporten kommer fokusera på skillnader och likheter mellan de tre språken C++, Prolog samt Lisp. Rapporten kommer enbart att fokusera de nyckelord och inbyggda funktioner som använts vid uppbyggnaden av Parsern och Lexern.

Vid en första anblick kan man se att språken inte alls är lika varandra men trots detta finns det likheter.

Inbyggd funktionalitet

C++ är inte känt för att ha mycket inbyggda funktioner i språket. Istället är mycket vanligt att man finner de flesta funktionerna i bibliotek. Exempel på detta är STL.

I Lisp och Prolog finns flera av de funktioner, som återfinns i C++'s bibliotek, redan inbydda i språket. Ett enkelt exempel på detta är hantering av strömmar. I C++ behövs biblioteket för att exmepelvis läsa och skriva till filer, skriva till skräm och även för att ta emot indata från användaren. I Prolog och Lisp är dessa funktioner inbyggda. Att funktionalitet är inbyggt har både för och nackdelar. Inbyggda funktioner ställer högre krav på kompilatorn medan biblioteken inte gör det.

Dessutom är det så att om mycket funktionalitet är inbyggt i språket blir valfriheten mindre för programmeraren. Om funktionalitet istället ligger i bibliotek kan dessa bytas ut eller göras om av programmeraren själv för att få exakt önskad funktionalitet.

Programmeringsstil

Under C++ finns det stor valfrihet när det gäller programmeringsstil. Det går mycket väl att programmera nästan helt och hållet objektorienterat, funktionellt eller iterativt.

I Prolog finns bara ett sätt att skriva programmen och det är att använda rekursion och följa den deduktiva apparaten som Prolog implementerar.

I Lisp, eftersom det är funktionellt, är hela programmet uppbyggt av funktionsanrop. Det finns möjlighet att, i CLisp, att skriva icke-funktionellt med hjälp av iterationer.

Felsökning

C++ har en mycket utbyggd mekanism för felsökning och den går att bygga ut ännu mer. När Parsern implementerades i C++ användes ingen debugger utan enbart den felinformation som erhölls från kompilatorn. Den information man erhåller från kompilatorn, i det här fallet GCC, är oftast mycket bra. Här finns information om var felet befinner sig och vad som är fel.

Prolog, till skillnad från C++, kompilerar inte någon programfil innan körning. Här sker detta direkt när filen läses in i kör-miljön. I detta skede visas fel som finns i programkoden. De fel som genereras från Prologs kompilator upplevdes, av programmerarna, som mycket mer svårtolkade. Detta är ett subjektivt mått och kan bero på programmerarnas kunskaper inom Prolog.

I Lsip är kompilatorn ganska lik Prolog. Felmeddelanden är inte alltid lika tydliga som i C++ och de flera av felen hittas inte förrän programmet exekveras. Måttet på felets tydlighet är även här mycket subjektivt och kan till stor del bestå i att djupare kunskap av Lisp fattas.

Körning av tester

Prolog och Lisp, tillskillnad från C++, har en egen miljö som programmet skall köras i. Detta begränsar användningen av ett program avsevärt. C++ kompilerar en exekverbar fil direkt till operativsystemets miljö medan Prolog och Lisp enbart tillåter körning från deras egen miljö.

Föra att köra alla de tester som behövs för att kontrollera Parserns funktionalitet skall ett antal filer läsas in från en media och därefter bearbeta dess innehåll genom Parsern.

Parsern som skrevs i C++ fungerar på det sätt att den exekverbara filen tar emot ett argument till den fil som skall Parsas. Detta innebär att man med ett enkelt shell-script kan köra genom samtliga testfiler genom att anropa programmet en gång för varje fil. Dessutom är det möjligt, med terminalens funktioner, att styra det så att utskriften från programmet hamnar i en text-fil på en media.

I Prolog fanns det inte, med information från det underlag som fanns tillgängligt, någon information om att det skulle vara möjligt att köra Parsern direkt från terminalen med

ett antal argument. Istället blev den enda möjligheten att göra en lista över samtliga filer som skulle Parsas. Genom rekursion gicks listan genom och för varje fil kördes Parsern. Innehållet från detta matades ut till körmiljön för Prolog. För att kunna samla information från alla testkörningar fick innehållet i körmiljön kopieras och sparas manuellt till en fil.

Lisp har, likt Prolog, en virtuell körmiljö som programmet måste köras inom. Detta begränsar, liksom för Prolog, möjligheten till att ta emot argument från terminalen. Det har dock framkommit i efterhand att detta har varit möjligt dock har denna möjlighet inte utnytjats. Istället används en inbyggd funktion i Lsip för att köra Parsern över en lista med filer. Här finns dock en stor fördel, i jämförelse med Prolog, och det är att det är möjligt att mata ut utskrifterna till en fil.

Slutsats

Den här rapporten kommer inte komma fram till vilken v de tre språken som är det bästa. Det är upp till läsaren att avgöra. Dock kommer ett par subjektiva bedömningar att presenteras nedan.

Fördelar och nackdelar

C++

C++ har den fördelen att det är ett Imperativt språk. Eftersom de flesta idag lär sig imperativa språk från grunden blir det lättast att förstår sig på detta och hur denna implementation fungerar. En annan fördel med C++ är att omfattningen av språket. Många använder i dag C++ vilket gör det till ett kraftigt språk med mycket att hämta. C++ har också fördelen av att det går att använda för att göra i stort sett allt inom datorteknik. På grund av C++s hårdvarunära implementation så kan koden göras väldigt resurseffektiv.

Nackdelen med C++ är ofta att mycket av det man vill göra är omständligt. Mycket skulle kunna vara enklare och mindre komplicerat. Dock skulle detta antagligen dra ner på språkets stora genomslagskraft och även dess funktionalitet. En annan stor nackdel med C++ är att den inte på långa vägar följer någon bra tankemodell för människor. Hur man än vrider och vänder på det hela så är det människor som programmerar datorprogram och då vore det mycket bra om språket var byggt så som människor tänker i allmänhet. Efter många års programmerande i C++ tar man över ett tänkande så som det fungerar men trots detta finns det brister. Här kan man främst se på det faktum att C++ har pekare. Hur pekare fungerar och vad pekare är går inte alls speciellt bra ihop med människans tänkande.

Om man ser till en mängden programkodsrader är C++ trots allt ovan värst. Här kräves mest rader skrivna för att åstadkomma något.

Prolog

Prolog har den stora fördelen att det är deklarativt. På så sätt slipper programmeraren att tala om exakt hur operationerna skall utföras, utan det räcker med att ange vad som skall göras. Detta ger att programmeraren kan koncentrera sig på vad som skall utföras istället för att ägna tid åt hur det skall utföras.

Prolog bygger på en deduktiv apparat men tyvärr tillåter inte dagens datorarkitekturer en perfekt implementation av detta. Istället arbetar prolog uppifrån och ner, sedan vänster till höger vilket ger att programmeraren ändå måste tänka på i vilken ordning som predikaten placeras. Det kan diskuteras ifall de är en för- eller nackdel att Prolog måste köras i en virtuell miljö. Dock ger oftast virtuella miljöer ett visst overhead och gör applikationerna avsevärt långsammare. Virtuella miljöer begränsar också möjligheten till att exekvera programmet. Den virtuella miljön, SWI-Prolog som har använts, begränsar också möjligheterna för omgivningen att styra Prolog-applikationen.

Prologs kräver enbart 99 programkodsrader vilket är exceptionellt utstående. Varje rad kan utföra mycket mer än i något av de andra två språken.

Lisp

Lisps stora fördel är att det klart och tydligt klarar av att överföra planering och idéer om hur programkod skall fungera direkt till hur det faktiskt skrivs. Tyvärr verkar Lisps fördelar sluta här igämförelse med de andra språken. Dock skall denna fördel inte underskattas då det är en stor konst att skriva bra datorprogram.

Lisp har samma nackdel som Prolog, så vida du inte har en Lisp-dator, att programmet måste exekvera i en virtuell miljö. Med samma resonemang som för Prolog så begränsar det här användandet av programmet avsevärt. En annan stor nackdel med Lisp är att programkoden är mycket svårläst. I det här fallet menas inte att programspråket i sig är svårt utan att det blir mycket moddigt med alla parenteser. Att skriva ett Lisp-program utan en textredigerare som kan stämma av parenteser ger mycket extraarbete med att enbart kontrollera så att samtliga parenteser stämmer. Det finns mycket här att göra för att öka läsbarheten men det är svårt att få det riktigt enkelt och snabbläst. Ofstast måste man räkna lite parenteser och jämföra vilka som är de olika argumenten för en funktion.

En annan tanke bakom hur Lisp fungerar är att det går implementra på samma sätt i C++. Detta ger att Lisp än en gång förlorar slagkraft mot de större och mer morderna språken. Lisp ligger dock mellan C++ och Prolog vad det gäller antalet programkodsrader vilket han ha en inverkan i valet av programspråk.

Bilagor

Bilaga A – Implementation i C++

Lexer.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <ctype.h>
#include <list>
#include <map>
#include <iomanip>
#include "tokenDef.h"
#include "tokenInfo.cpp"

#define LF  10
#define CR  13
#define TAB 9

using namespace std;
string programBuffer;

typedef  struct tab {
   char *text;
   int   token;
   } tab;

typedef struct symbolTable
{
   string role;
   string type;
   int size;
   int addr;

} symbolTable;

typedef struct operatorTable
{
   string opr;
   int arg1;
   int arg2;
   int result;

} operatorTable;

map<string,symbolTable> symbTable;
map<int,operatorTable> oprTable;

tab tokentab[] = {
             "id",       id,
             "number",   number,
             ":=",       assign,
             "",         0
             };

tab keywordtab[] = {
             "program",  program,
```

```
                "input",    input,
                "output",   output,
                "var",      var,
                "begin",    begin_,
                "end",      end_,
                "integer",  integer,
                "boolean",  boolean,
                "real",     real_,
                "",         0
                };

//Add the std datatypes to the ST
void fillSymbTable()
{
   symbolTable inten;
   inten.role = "type";
   inten.type = "_predef";
   inten.size = 4;
   inten.addr = 0;
   symbTable["integer"] = inten;

   symbolTable boolen;
   boolen.role = "type";
   boolen.type = "_predef";
   boolen.size = 1;
   boolen.addr = 0;
   symbTable["boolean"] = boolen;

   symbolTable realen;
   realen.role = "type";
   realen.type = "_predef";
   realen.size = 16;
   realen.addr = 0;
   symbTable["real"] = realen;

}

//Add information to the operation table.
void fillOprTable()
{
   operatorTable toAdd;

   toAdd.opr = "+";
   toAdd.arg1 = integer;
   toAdd.arg2 = integer;
   toAdd.result = integer;
   oprTable[1] = toAdd;

   toAdd.opr = "+";
   toAdd.arg1 = real_;
   toAdd.arg2 = integer;
   toAdd.result = real_;
   oprTable[2] = toAdd;

   toAdd.opr = "+";
   toAdd.arg1 = integer;
   toAdd.arg2 = real_;
   toAdd.result = real_;
   oprTable[3] = toAdd;
```

```
toAdd.opr = "+";
toAdd.arg1 = real_;
toAdd.arg2 = real_;
toAdd.result = real_;
oprTable[4] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = integer;
toAdd.arg2 = integer;
toAdd.result = integer;
oprTable[5] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = real_;
toAdd.arg2 = integer;
toAdd.result = real_;
oprTable[6] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = integer;
toAdd.arg2 = real_;
toAdd.result = real_;
oprTable[7] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = real_;
toAdd.arg2 = real_;
toAdd.result = real_;
oprTable[8] = toAdd;

toAdd.opr = "+";
toAdd.arg1 = integer;
toAdd.arg2 = number;
toAdd.result = integer;
oprTable[9] = toAdd;

toAdd.opr = "+";
toAdd.arg1 = number;
toAdd.arg2 = integer;
toAdd.result = integer;
oprTable[10] = toAdd;

toAdd.opr = "+";
toAdd.arg1 = number;
toAdd.arg2 = integer;
toAdd.result = real_;
oprTable[11] = toAdd;

toAdd.opr = "+";
toAdd.arg1 = integer;
toAdd.arg2 = number;
toAdd.result = real_;
oprTable[12] = toAdd;

toAdd.opr = "+";
toAdd.arg1 = real_;
toAdd.arg2 = number;
toAdd.result = real_;
oprTable[13] = toAdd;
```

```
toAdd.opr = "+";
toAdd.arg1 = number;
toAdd.arg2 = real_;
toAdd.result = real_;
oprTable[14] = toAdd;

toAdd.opr = "+";
toAdd.arg1 = number;
toAdd.arg2 = number;
toAdd.result = number;
oprTable[15] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = integer;
toAdd.arg2 = number;
toAdd.result = integer;
oprTable[16] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = number;
toAdd.arg2 = integer;
toAdd.result = integer;
oprTable[17] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = number;
toAdd.arg2 = integer;
toAdd.result = real_;
oprTable[18] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = integer;
toAdd.arg2 = number;
toAdd.result = real_;
oprTable[19] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = real_;
toAdd.arg2 = number;
toAdd.result = real_;
oprTable[20] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = number;
toAdd.arg2 = real_;
toAdd.result = real_;
oprTable[21] = toAdd;

toAdd.opr = "*";
toAdd.arg1 = number;
toAdd.arg2 = number;
toAdd.result = number;
oprTable[22] = toAdd;

toAdd.opr = ":=";
toAdd.arg1 = real_;
toAdd.arg2 = unknown;
toAdd.result = real_;
oprTable[24] = toAdd;
```

```
   toAdd.opr = ":=";
   toAdd.arg1 = integer;
   toAdd.arg2 = unknown;
   toAdd.result = integer;
   oprTable[25] = toAdd;

   toAdd.opr = ":=";
   toAdd.arg1 = boolean;
   toAdd.arg2 = unknown;
   toAdd.result = boolean;
   oprTable[26] = toAdd;

}

void printSymbolTable()
{

  map<string,symbolTable>::iterator it;

  cout << endl << "                    Symbol Table";
  cout << endl << "                    ============" << endl << endl;

  cout << setiosflags(ios::left) << setw(20) << "Name" << setw(10)
       << "Role" << setw(10) << "Type" << setw(5) << "Size" <<
setw(10) << "Address" << endl << "-----------------------------------
------------------" << endl;

  // show content:
  for (it=symbTable.begin(); it != symbTable.end(); it++ )
    cout << setiosflags(ios::left) << setw(20) << (*it).first <<
setw(10) << (*it).second.role << setw(10) << (*it).second.type <<
setw(5) << (*it).second.size << setw(10) << (*it).second.addr <<
endl;

   cout << endl << endl;

}

void printOprTable()
{

  map<int,operatorTable>::iterator it;

  cout << endl << "                    Operator Table";
  cout << endl << "                       ==============" << endl <<
endl;

  cout << setiosflags(ios::left) << setw(10) << "Operator" <<
setw(10)
       << "Arg1" << setw(10) << "Arg2" << setw(10) << "Result" <<
endl << "-------------------------------------------------" <<
endl;

  // show content:
  for (it=oprTable.begin(); it != oprTable.end(); it++ )
    cout << setiosflags(ios::left) << setw(10) << (*it).second.opr
<< setw(10) << (*it).second.arg1 << setw(10) << (*it).second.arg2 <<
setw(10) << (*it).second.result << endl;

   cout << endl << endl;
```

```
}

//Read the program source code from the filebuffer
bool readInProgram(char fileName[], string& programText)
{
   string line;
   ifstream source(fileName);

   if (source.is_open())
   {
      while(!source.eof())
      {
         getline(source,line);
         programText.append("\t");
         programText.append(line);
         programText.append("\n");
      }
      source.close();

      return true;
   }
   else
   {
      cout << "*** File '" << fileName << "' not found or file
corrupt..." << endl << endl;
      return false;
   }

}

//Match a keyword in the string at position with one in the
keywordtable
int matchKeyword(string &word, int position)
{
   if(keywordtab[position].token == 0 || keywordtab[position].text
== word)
      return keywordtab[position].token;
   else
      matchKeyword(word, position + 1);
}

//Get the length of a alfanumric word in a string, starting at
position
int getWordLen(string &buffer, int position)
{
   return isalpha(buffer[position]) || isdigit(buffer[position]) ?
getWordLen(buffer, position + 1) : position;
}

//Get the length of a number in a string, startinga at position
int getNumLen(string &buffer, int position)
{
   return isdigit(buffer[position]) ? getNumLen(buffer, position +
1) : position;
}

//Checks if an expression has the correct datatypes
bool checkExpr(list<string> &expr)
{
```

```
   bool wentOk = true;
   string a, b, c;
   int pre, cType;

   a = expr.front(); expr.pop_front();

   if(symbTable.find(a) != symbTable.end())
      pre = matchKeyword(symbTable[a].type,0);
   else if(a == "number")
      pre = integer;
   else
      pre = unknown;

   while(expr.size() >= 2)
   {
      b = expr.front(); expr.pop_front();
      c = expr.front(); expr.pop_front();
      map<int,operatorTable>::iterator oprIt;

      if(symbTable.find(c) != symbTable.end())
      {
         cType = matchKeyword(symbTable[c].type,0);
         if(cType == 0)
           cType = unknown;
      }
      else if(c == "number")
         cType = integer;
      else
         cType = unknown;

      bool found = false;

      if( b == ":=")
      {
         for(oprIt=oprTable.begin();  oprIt  !=  oprTable.end()  &&
!found; oprIt++ )
           if((*oprIt).second.opr == b)
           {
             if((*oprIt).second.arg1 == pre && (*oprIt).second.result
== cType)
                found = true;
           }
      }
      else
      {
         for  (oprIt=oprTable.begin();  oprIt  !=  oprTable.end()  &&
!found; oprIt++ )
           if((*oprIt).second.opr == b)
           {
             if((*oprIt).second.arg1  ==  pre  &&  (*oprIt).second.arg2
== cType)
             {
                pre = (*oprIt).second.result;
                found = true;
             }
           }
      }

      if(!found)
      {
```

```
         cout << endl << "*** Mismatching types. Variable " << c <<
" of type ";
         coutTokenInfo(cType);
         cout << endl << "     is not compatible with expression of
type ";
         coutTokenInfo(pre);
         pre = unknown;
         wentOk = false;
      }
   }

   return wentOk;
}

//Returns a token and information about if getToken was sucessfully
to the parser, it does also add symbols to the ST
pair<int,bool> getTokenArg(string &buffer)
{
   static int position = 0;
   static int pgmPart;
   static string pgmName;
   static string varType;
   static list<string> unSetVar;
   static int addrOffset = 0;
   static list<string> expr;
   static bool endFound = false;
   int lexLen = 0, token;
   bool wentOk = true;
   string tempBuffer;
   pair<map<string,symbolTable>::iterator,bool> ret;

   if(isalpha(buffer[position]))
   {
      lexLen = getWordLen(buffer,position) - position;
      tempBuffer = buffer.substr(position,lexLen);
      position += lexLen;
      token = matchKeyword(tempBuffer,0);

      switch(token)
      {
         case program: pgmPart = program; break;
         case var: pgmPart = var; break;
         case begin_: pgmPart = begin_; break;
         case end_: pgmPart = end_; break;

      };

      if(!token)
      {
         symbolTable add;

         if(pgmPart == program)
         {
           pgmName = tempBuffer;
           add.role = "program";
           add.type = "_predef";
           add.size = 0;
           add.addr = 0;
         }
         if(pgmPart == var)
```

```
          {
            add.role = "variable";
            add.type = "unknown";
            add.size = 0;
            add.addr = 0;
            unSetVar.push_front(tempBuffer);
          }
          if(pgmPart == unknown)
          {
            add.role = "unknown";
            add.type = "unknown";
            add.size = 0;
            add.addr = 0;
          }

          if(pgmPart == var || pgmPart == program || pgmPart ==
unknown)
          {
            ret                    =                    symbTable.insert
(pair<string,symbolTable>(tempBuffer,add));
            if(ret.second == false)
            {
              cout << endl << endl << "*** Variable " << tempBuffer <<
" already defined." <<  endl;
              if(unSetVar.size() > 0)
                 unSetVar.pop_back();
              wentOk = false;
            }

          }

          if(pgmPart == begin_)
          {
            if(symbTable.find(tempBuffer) == symbTable.end())
            {
              cout << endl  << endl << "*** Variable " << tempBuffer
<< " is not defined." <<  endl;
              wentOk = false;
            }

            if(!token)
              expr.push_front(tempBuffer);
          }

          if(pgmPart == program)
            pgmPart = unknown;
       }

       if(token == integer || token == boolean || token == real_)
       {

          while(!unSetVar.empty())
          {
            string currentVar = unSetVar.back();
            unSetVar.pop_back();
            varType = tempBuffer;

            symbTable[currentVar].size = symbTable[varType].size;
            symbTable[currentVar].addr = addrOffset;
            symbTable[currentVar].type = varType;
```

```
            addrOffset += symbTable[varType].size;


            if(symbTable.find(pgmName) != symbTable.end())
              symbTable[pgmName].size = addrOffset;

          }

      }
      pair<int, bool> retur((token ? token : id),wentOk);
      return retur;
   }
   else if(isdigit(buffer[position]))
   {
      lexLen = getNumLen(buffer,position) - position;
      expr.push_front("number");
      position += lexLen;
      pair<int, bool> retur(number,wentOk);
      return retur;
   }
   else
   {
      int tempPos = position;

      if(pgmPart == begin_ && expr.size() > 2 && (buffer[tempPos] ==
CR || buffer[tempPos] == LF || buffer[tempPos] == ';'))
         wentOk = checkExpr(expr);
      else   if(pgmPart   ==   begin_   &&   expr.size()   <=   2   &&
(buffer[tempPos] == CR || buffer[tempPos] == LF || buffer[tempPos]
== ';'))
         expr.clear();

      if(buffer[tempPos] == ':' && buffer[tempPos+1] == '=')
      {
         position += 2;
         expr.push_front(":=");
         pair<int, bool> retur(assign,wentOk);
         return retur;
      }
      else if(buffer[tempPos] == ' ' || buffer[tempPos] == CR ||
buffer[tempPos] == LF || buffer[tempPos] == TAB )
      {
         position += 1;
         pair<int,bool> returen;
         returen = getTokenArg(buffer);
         if(returen.second == false)
           wentOk = false;
         pair<int, bool> retur(returen.first,wentOk);
         return retur;
      }
      else
      {
         if(buffer[position] == '+' || buffer[position] == '*')
         {
           string toInsert;
           toInsert = buffer.substr(position,1);
           expr.push_front(toInsert);
         }
         if(buffer[tempPos] != '$')
```

```
            position += 1;

         pair<int, bool> retur(buffer[tempPos],wentOk);
         return retur;
      }
   }
}

//The lexer was originally built without non-local dataobjekts
//Insted of re-writing the orginal function we just wrap it in.
pair<int,bool> getToken()
{
   return getTokenArg(programBuffer);
}
Parser.cpp
#include <iostream>
#include <cstring>
#include "lexer.cpp"
#include "tokenDef.h"
#include "tokenInfo.cpp"
#define DEBUG false

using namespace std;

int lookahead;

//Matches a expected token with the next token in the program
bool match(int token)
{

   pair<int,bool> returen;

   if(DEBUG)
   {
      cout << endl << "*   In 'match'. Token infomration; expected:
";
         coutTokenInfo(token);
      cout << ", found: ";
      coutTokenInfo(lookahead);
   }

   if (lookahead == token)
   {
      returen = getToken();
      lookahead = returen.first;
      return returen.second;
   }
      else
      {
         cout << endl << "*** Unexpected Token: expected:  ";
         coutTokenInfo(token);
      cout << endl << "                        found:     ";
      coutTokenInfo(lookahead);
      //returen = getToken();
      //lookahead = returen.first;
         return false;
      }

}
```

```
//Print an error message
bool printError(string tokenInfo)
{
   pair<int,bool> returen;

   if(DEBUG)
      cout << endl << "*   In 'printError'.";

   cout << endl << "*** Unexpected Token: expected:  ";
   cout << tokenInfo;
   cout << endl << "                       found:     ";
   coutTokenInfo(lookahead);
   returen = getToken();
   lookahead = returen.first;
      return false;
}

// [prog header] ::= program id ( input , output );
bool programHeader()
{
   if(DEBUG)
      cout << endl << "*   In 'programHeader'";

   return (match(program) & match(id) & match('(') & match(input) &
match(',') & match(output) & match(')') & match(';')) ? true :
false;
}

//[id list] ::= id | [id list] , id
bool idList()
{
   if(DEBUG)
      cout << endl << "*   In 'idList'";

   bool returnVal1 = true, returnVal2 = true;

   returnVal1 = match(id);
   if(lookahead == ',')
      returnVal2 = match(',') & idList();

   return returnVal1 & returnVal2;
}

//[type] ::= integer | boolean
bool type()
{
   if(DEBUG)
      cout << endl << "*   In 'type'";

   if(lookahead == integer)
      return match(integer);
   else if(lookahead == boolean)
      return match(boolean);
   else if(lookahead == real_)
      return match(real_);
   else
      return printError("type"); //Print error message and read next
token

}
```

```
//[var dec] ::= [id list] : [type]  ;
bool varDec()
{
   if(DEBUG)
      cout << endl << "*   In 'varDec'";

   return (idList() & match(':') & type() & match(';')) ? true :
false;
}

//[var dec list] ::= [var dec] | [var dec list] [var dec]
bool varDecList()
{
   if(DEBUG)
      cout << endl << "*   In 'varDecList'";

   bool returnVal1 = true, returnVal2 = true;

   returnVal1 = varDec();
   if(lookahead != begin_ && returnVal1)
      returnVal2 = varDecList();

   return returnVal1 & returnVal2;
}

//[var part] ::= var [var dec list]
bool varPart()
{
   if(DEBUG)
      cout << endl << "*   In 'varPart'";

   return match(var) & varDecList();
}

//[operand] ::= id | number
bool operand()
{
   if(DEBUG)
      cout << endl << "*   In 'operand'";

   if(lookahead == id)
      return match(id);
   else if(lookahead == number)
      return match(number);
   else
      return printError("operand");
}

//[operator] ::=  + | *
bool operatorn()
{

   if(DEBUG)
      cout << endl << "*   In 'operatorn'";

   if(lookahead == '*')
      return match('*');
   else if(lookahead == '+')
      return match('+');
```

```
   else
      return printError("operator");

}

//[expr] ::=  [operand] | [expr] [operator] [operand]
bool expr()
{
   if(DEBUG)
      cout << endl << "*   In 'expr'";

   bool returnVal1 = true, returnVal2 = true;

   returnVal1 = operand();
   if(lookahead == '*' || lookahead == '+')
      returnVal2 = operatorn() & expr();

   return returnVal1 & returnVal2;
}

//[assign stat] ::= id := [expr]
bool assignStat()
{
   if(DEBUG)
      cout << endl << "*   In 'assignStat'";

   return match(id) & match(assign) & expr();
}

//[stat] ::= [assign stat]
bool stat()
{
   if(DEBUG)
      cout << endl << "*   In 'stat'";

   return assignStat();
}

//[stat list] ::= [stat] | [stat list]  ;  [stat]
bool statList()
{
   if(DEBUG)
      cout << endl << "*   In 'statList'";

   bool returnVal1 = true, returnVal2 = true;

   returnVal1 = stat();
   if(lookahead == ';')
      returnVal2 =  match(';') & statList();

   return returnVal1 & returnVal1;
}

//Check if there is any chars left
bool leftBuffer()
{
   if(DEBUG)
      cout << endl << "*   In 'leftBuffer'";

   if(lookahead == '$')
```

```
      return true;
   else
   {
      cout << endl << "*** Extra symbols after end of parse";
      return false;
   }
}

//[stat part] ::= begin [stat list] end .
bool statPart()
{
   if(DEBUG)
      cout << endl << "*   In 'statPart'";

   if(match(begin_) & statList() & match(end_) & match('.'))
      return leftBuffer();
   else
      return false;
}

//[prog] ::=  [prog header] [var part] [stat part]
bool prog()
{
   pair<int,bool> returen;

   if(DEBUG)
      cout << endl << "*   In 'prog'";

   //Fill the Operator-table
   fillOprTable();

   //Fill the symbol table
   fillSymbTable();

   returen = getToken();

   lookahead = returen.first;

   return programHeader() & varPart() & statPart() & returen.second;
}

//Run the program, read the input data and read the fil in to the
buffer
//Start the parser
int main(int argc, char *argv[])
{
   if(argc == 2)
   {
      int token;
      if(readInProgram(argv[1], programBuffer))
      {
         cout << "*    --- Program '" << argv[1] << "' is ---" <<
endl << endl;
         cout << programBuffer << endl;
         programBuffer.append("$");
         cout << "*   --- Start parsing ---" << endl;
         prog() ? (cout << endl << endl << "*    --------- Parse
Successful! ---------   *" << endl << endl) : (cout << endl << endl
<< "*   --------- Parse failed! ---------   *" << endl << endl);
         printSymbolTable();
```

```
         //printOprTable();
      }
   }
   else
      cout << "*** Wrong number of arguments supplied..." << endl <<
"*   Usage: " << argv[0] << " file_to_check" << endl << endl;


   return 0;

}
```

tokenDef.h

```
#ifndef _tokenDEF1234567890_
#define _tokenDEF1234567890_

   #define start     256
   #define id        start    +1
   #define number    id       +1
   #define assign    number   +1

   #define program   assign   +1
   #define input     program  +1
   #define output    input    +1
   #define var       output   +1
   #define begin_    var      +1
   #define end_      begin_   +1
   #define integer   end_     +1
   #define boolean   integer  +1
   #define unknown    boolean  +1
   #define real_   unknown  +1

#endif
```

tokenInfo.h

```
#ifndef _tokenInfoDEF1234567890_
#define _tokenInfoDEF1234567890_

void coutTokenInfo(int ascii)
{
   using namespace std;

   if(ascii == '$')
      cout << "EOF";
   else if(ascii < start)
      cout << ((char) ascii);
   else
      {
         switch (ascii)
         {

           case start:      cout << "start";     break;
           case id:      cout << "id";        break;
           case number:  cout << "number";       break;
           case assign:  cout << ":=";        break;
           case program:    cout << "program";       break;
           case input:   cout << "input";     break;
           case output:  cout << "output";       break;
```

```
            case var:     cout << "var";          break;
            case begin_:  cout << "begin";        break;
            case end_:       cout << "end";           break;
            case integer:    cout << "integer";       break;
            case boolean:    cout << "boolean";       break;
            case unknown:    cout << "unknown";       break;
            case real_:   cout << "real";      break;
            default:      cout << "unknown";      break;
          };

      }

}


#endif
```

Bilaga B – Implementation i Prolog

```
/* TESTER ATT KORA */

runProgram(_):-
parseFiles(['Test/testok1.pas','Test/testok2.pas','Test/testok3.pas'
,'Test/testok4.pas','Test/testok5.pas','Test/testok6.pas','Test/test
ok7.pas','Test/fun1.pas',      'Test/fun2.pas',      'Test/fun3.pas',
'Test/fun4.pas',          'Test/fun5.pas',          'Test/testa.pas',
'Test/testb.pas',         'Test/testc.pas',         'Test/testd.pas',
'Test/teste.pas',         'Test/testf.pas',         'Test/testg.pas',
'Test/testh.pas',         'Test/testi.pas',         'Test/testj.pas',
'Test/testk.pas',         'Test/testl.pas',         'Test/testm.pas',
'Test/testn.pas',         'Test/testo.pas',         'Test/testp.pas',
'Test/testq.pas',         'Test/testr.pas',         'Test/tests.pas',
'Test/testt.pas',         'Test/testu.pas',         'Test/testv.pas',
'Test/testw.pas',         'Test/testx.pas',         'Test/testy.pas',
'Test/testz.pas'        'Test/sem4.pas',         'Test/sem5.pas']).

parseFiles([]).
parseFiles([H|T]):-parse(H), write('\n\n'), parseFiles(T).


/* -- LEXER -- */
singleCharacter(44). /*,*/
singleCharacter(59). /*;*/
singleCharacter(58). /*:*/
singleCharacter(63). /*?*/
singleCharacter(33). /*!*/
singleCharacter(46). /*.*/
singleCharacter(40). /*(*/
singleCharacter(41). /*)*/
singleCharacter(42). /***/
singleCharacter(43). /*+*/
singleCharacter(61). /*=*/
singleCharacter(-1). /*EOF*/

keyWord(program).
keyWord(input).
keyWord(output).
keyWord(var).
keyWord(begin).
```

```
keyWord(end).
keyWord(integer).
keyWord(real).

translateToToken([:,=],assign):-!.
translateToToken(.,.):-!.
translateToToken((,),,):-!.
translateToToken('(','('):-!.
translateToToken(')',')'):-!.
translateToToken(:,:):-!.
translateToToken(;,;):-!.
translateToToken(*,*):-!.
translateToToken(+,+):-!.
translateToToken(-1,eof):-!.
translateToToken(S,number):-S\=[_,_],      atom_chars(S,      [H|_]),
isNum(H).
translateToToken(L,id):-L\=[_,_],!.

inWord(C,C):-C>96, C<123. /* a - z */
inWord(C,L):-C>64, C<91, L is C+32. /*A - Z*/
inWord(C,C):-C>47, C<58. /*1-9*/
inWord(39,39). /*' (')*/
inWord(45,45). /*-*/

lastWord('.').
lastWord(-1).

isNum(0).
isNum('0').
isNum(1).
isNum('1').
isNum(2).
isNum('2').
isNum(3).
isNum('3').
isNum(4).
isNum('4').
isNum(5).
isNum('5').
isNum(6).
isNum('6').
isNum(7).
isNum('7').
isNum(8).
isNum('8').
isNum(9).
isNum('9').

openFile(File,S):-see(File), readIn(S), seen.

readIn([W|Ws]):-get0(C), readWord(C,W,C1), restSent(W,C1,Ws).

restSent(W,_,[]):- lastWord(W), !.
restSent(_,C,[W1|Ws]):-readWord(C,W1,C1), restSent(W1,C1,Ws).

readWord(C,W,C1):-C\=(-1),   singleCharacter(C),   !,   name(W,[C]),
get0(C1).
readWord(C,W,C2):-C\=(-1),      inWord(C,NewC),      !,      get0(C1),
restWord(C1,Cs,C2), name(W,[NewC|Cs]).
readWord(C,W,C2):-C\=(-1), get0(C1), readWord(C1,W,C2).
```

```
readWord(_,-1,-1):-!.

restWord(C,[NewC|Cs],C2):-inWord(C,NewC),        !,        get0(C1),
restWord(C1,Cs,C2).
restWord(C,[],C).

makeTokenList([E],E):-keyWord(E), !.
makeTokenList([E],[B]):-translateToToken(E,B), !.
makeTokenList([H|T],[H|B]):-keyWord(H), !, makeTokenList(T,B).
makeTokenList([H,H2|T],[F|B]):-translateToToken([H,H2],F),        !,
makeTokenList(T,B).
makeTokenList([H|T],[F|B]):-translateToToken(H,F),
makeTokenList(T,B).

getTokensFromFile(File, B):-openFile(File, S), makeTokenList(S,B).


/* -- PARSER -- */
match(To, [To|Tail], Tail):-!.

parse(File):-getTokensFromFile(File,B),   write('--   Program:   '),
write(File), write(' --\n'), prog(B).

prog(P):-progHeader(P,T),  varPart(T,Y),  statPart(Y),  !,  write('**
Parse successful.\n').
prog(_):-write('** Parse failed.\n').

progHeader(In,Ut):-match(program,In,Sv1),
            match(id,Sv1,Sv2),
            match('(',Sv2,Sv3),
            match(input,Sv3,Sv4),
            match(,,Sv4,Sv5),
            match(output,Sv5,Sv6),
            match(')',Sv6,Sv7),
            match(;,Sv7,Ut).

varPart(In,Ut):-   match(var,In,Sv1),
            varDecList(Sv1,Ut).

varDecList(In,     Ut):-varDec(In,     [H|T]),     H\=begin,     !,
varDecList([H|T], Ut).
varDecList(In, Ut):-varDec(In, Ut).

varDec(In,   Ut):-          idList(In,   Sv1),   match(:,Sv1,Sv2),
varType(Sv2,Sv3), match(;,Sv3,Ut).

idList(In, Ut):-    match(id, In, [H|T]), H=,, !, match(,,[H|T],
Sv1), idList(Sv1,Ut).
idList(In, Ut):-   match(id, In, Ut).

varType(In, Ut):-  match(integer,In, Ut);match(real, In, Ut).

statPart(In):-           match(begin,In,Sv1),  statList(Sv1,  Sv2),
match(end, Sv2, Sv3), match(.,Sv3,Rest), Rest=[].

statList(In,   Ut):-   stat(In,   [H|T]),   H=;,   match(;,[H|T],Sv1),
statList(Sv1, Ut).
statList(In, Ut):- stat(In, Ut).

stat(In, Ut):-     assignStat(In, Ut).
```

```
assignStat(In,  Ut):-match(id,In,Sv1),  match(assign,  Sv1,  Sv2),
expr(Sv2, Ut).

expr(In, Ut):-     operand(In,     [H|T]),     (H='*';     H='+'),
operator([H|T], Sv2), expr(Sv2, Ut).
expr(In, Ut):-     operand(In, Ut).

operand(In, Ut):-  match(id, In, Ut); match(number, In, Ut).

operator(In, Ut):- match(+, In, Ut); match(*, In, Ut).
```

Bilaga C – Implementation i Lisp

```
;=========================================================
; Denna funktion laeser fraan stream. Extraherar ett lexeme.
; skippa white-space tecken
;=========================================================

(defun stream-collect-lexeme (stream ws-characters)

  (when (not (eq 'EOF (loop
            (let ((c (peek-char nil stream nil 'EOF)))
       (if (or (eq c 'EOF) (not (member c ws-characters)))
           (return c)
         (read-char stream))))))
    (let ((lexeme ""))
      (loop
       (let ((c (peek-char nil stream nil 'EOF)))
    (if (or (eq c 'EOF) (member c ws-characters))
        (return lexeme)
      (progn
        (read-char stream)
        ; Gah! Var fan finns string-append-char!??!?!!?
        ; Taenk: (setf lexeme (string-append-char lexeme c))
        (setf lexeme
       (concatenate 'string lexeme
            (make-string 1 :initial-element c)))))))))

;=========================================================
; Returvaerde fraan get-next-token aer alltid en lista.
; car  returvaerde = Token   (ex: 'IF ':= '< osv.)
; cadr returvaerde = Lexeme  (ex: "IF"  ":=" "<")
;=========================================================

(defun get-next-token (stream lexeme-mapper)
  (let  ((result  (stream-collect-lexeme  stream  '(#\Space  #\Tab
#\Newline))))
    (if (null result) (list 'EOF "")(funcall lexeme-mapper result))
  )
 )

;=========================================================
;Haer skapas en lista innehaallande tvaa element:
;(symbol, lexeme)
;=========================================================

(defun map-lexeme (lexeme)
```

```
;(format t "The incoming lexeme is: ~S ~%" lexeme)
   (list (cond

      ((string= lexeme "program")    'PROGRAM)
      ((string= lexeme "var")        'VAR)
      ((string= lexeme "input")      'INPUT)
      ((string= lexeme "output")  'OUTPUT)
     ((string= lexeme ":=")         'ASSIGN)
      ((string= lexeme ",")          'COMMA)
      ((string= lexeme ";")          'SCOLON)
      ((string= lexeme ":")          'COLON)
      ((string= lexeme "(")          'LP)
      ((string= lexeme ")")          'RP)
     ((string= lexeme "+")          'ADD)
     ((string= lexeme "*")          'TIMES)
      ((string= lexeme "integer")    'INTEGER)
      ((string= lexeme "real")       'REAL)
     ((string= lexeme "begin")      'BEGIN)
     ((string= lexeme ".")        'DOT)
     ((string= lexeme "end")        'END)

      ((is-id lexeme)                'ID)
      ((is-number lexeme)            'NUMBER)

      (t                             'UNKNOWN)
      )
    lexeme)
)

;============================================================
;Jaemfoer om foersta tecknet i straengen aer en bokstav
;samt att resterande tillhoer maengden {a-z, A-Z, 0-9}
;============================================================

(defun is-id (str)
   (and (alpha-char-p (char str 0))
       (every #'is-alphanum str)
   )
)

(defun is-alphanum(char)
   (cond
      ((digit-char-p char)   t)
      ((alpha-char-p char)   t)
      (t             nil)
   )
)


(defun is-number (str)
   (every #'digit-char-p str)
)

;============================================================
;Skapar en strukt med tvaa faelt:
;lookahead aer en lista (symbol, lexeme), stream aer en
;filstroem.
;============================================================

(defstruct pstate
```

```
    (lookahead)
    (stream)
)

;=========================================================
;Konstruktor foer strukten pstate, initialiserar
;stream till inskickad filstroem och lookahead
;till foersta inlaesta token (symbol, lexeme)
;=========================================================

(defun create-parser-state (stream)

    (make-pstate :stream stream
                 :lookahead (get-next-token stream #'map-lexeme)
    )

)

;=========================================================
;match haemtar ett nytt token fraan filstroemmen om
;inskickad symbol matchar lookahead, annars har det
;uppstaatt ett syntax error.
;=========================================================

(defun match (state symbol)

    (if (eq symbol (first (pstate-lookahead state)))
        (setf (pstate-lookahead state)
            (get-next-token (pstate-stream state) #'map-lexeme)
        )
        (format t "*** Expected: ~S, found: ~D. ~%~%" symbol (first
(pstate-lookahead state)))
    )

)


(defun matchNoOutput (state symbol)
    (if (eq symbol (first (pstate-lookahead state)))
        (setf (pstate-lookahead state)
            (get-next-token (pstate-stream state) #'map-lexeme)
        )
    )
)

;=========================================================
;Startat programmet genom att man vid promten skriver:
;
;(parser)
;
;Programmet kommer daa att fraaga efter ett filnamn, finns
;filen kommer den att bli parserad och ett meddelande
;kommer att skrivas ut som meddelar om parsningen gick bra.
;=========================================================

(defun parser (file)
   (format t "--- Parsing file: ~S --- ~%" file)
    (with-open-file (stream file :direction :input)
       (if  (program (create-parser-state stream))
```

```
                    (format t "*   Computer says yes!  (Parser says
true). ~%~%")
                    (format t "*   Computer says no!   (Parser says
false). ~%~%")
       )
    )
)

;=============================================================
;[prog header] ::= program id ( input , output );
;=============================================================
(defun progHeader(state)

   (and (match state 'PROGRAM)
        (match state 'ID)
        (match state 'LP)
        (match state 'input)
        (match state 'COMMA)
        (match state 'output)
        (match state 'RP)
        (match state 'SCOLON)
   )

)

;=============================================================
;[var part] ::= var [var dec list]
;=============================================================
(defun varPart(state)

   (and (match state 'VAR)
        (varDecList state)
   )
)

;=============================================================
;[var dec list] ::= [var dec] | [var dec list] [var dec]
;=============================================================
(defun varDecList(state)

   (and (varDec state)
        (if (not (eq 'BEGIN (first (pstate-lookahead state))))
          (VarDecList state)
          t
        )
   )
)

;=============================================================
;[var dec]  ::= [id list] : [type]  ;
;=============================================================
(defun varDec(state)

   (and (idList state)
       (match state 'COLON)
       (typeEn state)
       (match state 'SCOLON)
   )
)
```

```
;=============================================================
;[id list] ::= id | [id list] , id
;=============================================================
(defun idList(state)

   (and (match state 'ID)
        (if (eq 'COMMA (first (pstate-lookahead state)))
          (and (match state 'COMMA)
               (idList state)
          )
          t
        )
   )
)

;=============================================================
;[type] ::= integer | real
;=============================================================
(defun typeEn(state)

   (if (or   (matchNoOutput state 'INTEGER)
         (matchNoOutput state 'REAL))
      t
      (format t "*** Expected: ~S, found: ~D. ~%~%" 'TYPE (first
(pstate-lookahead state)))

   )
)

;=============================================================
;[stat part] ::= begin [stat list] end .
;=============================================================
(defun statPart(state)

   (and (match state 'BEGIN)
        (statList state)
        (match state 'END)
        (match state 'DOT)
   )

)

;=============================================================
;[stat list] ::= [stat] | [stat];  [stat list]
;=============================================================
(defun statList(state)

   (and (stat state)
        (if (eq 'SCOLON (first (pstate-lookahead state)))
          (and    (match state 'SCOLON)
                  (statList state)
          )

          t
        )
   )
)

;=============================================================
;[stat] ::= [assign stat]
```

```
;=========================================================
(defun stat(state)

   (assignStat state)
)

;=========================================================
;[assign stat] ::= id := [expr]
;=========================================================
(defun assignStat(state)

   (and (match state 'ID)
        (match state 'ASSIGN)
        (expr state)
        )
)

;=========================================================
;[expr] ::=  [operand] | [operand] [operator] [expr]
;=========================================================
(defun expr(state)

   (and (operand state)
       (if  (or  (eq 'TIMES (first (pstate-lookahead state)))
                 (eq 'ADD (first (pstate-lookahead state)))
            )
         (and (operatorn state)
           (expr state)
         )
         t
       )
   )
)

;=========================================================
;[operand]  ::= id | number
;=========================================================
(defun operand(state)

   (if (or (matchNoOutput state 'ID)
        (matchNoOutput state 'NUMBER))
      t
      (format t "*** Expected: ~S, found: ~D. ~%~%" 'OPERAND (first
(pstate-lookahead state)))
   )

)
;=========================================================
;[operator] ::=  + | *
;=========================================================
(defun operatorn(state)


   (if (or (matchNoOutput state 'ADD)
           (matchNoOutput state 'TIMES))
      t
      (format t "*** Expected: ~S, found: ~D. ~%~%" 'OPERATOR (first
(pstate-lookahead state)))

   )
```

```
)

;=========================================================
;<program> --> <program-header><var-part><stat-part>EOF
;=========================================================

(defun program (state)

   (and (progHeader state)
      (varPart state)
      (statPart state)
        (match state 'EOF)
   )

)

(defun start()

   (setf filer '(Test/testok1.pas Test/testok2.pas Test/testok3.pas
Test/testok4.pas Test/testok5.pas Test/testok6.pas Test/testok7.pas
Test/fun1.pas     Test/fun2.pas     Test/fun3.pas     Test/fun4.pas
Test/fun5.pas    Test/testa.pas    Test/testb.pas    Test/testc.pas
Test/testd.pas    Test/teste.pas    Test/testf.pas    Test/testg.pas
Test/testh.pas    Test/testi.pas    Test/testj.pas    Test/testk.pas
Test/testl.pas    Test/testm.pas    Test/testn.pas    Test/testo.pas
Test/testp.pas    Test/testq.pas    Test/testr.pas    Test/tests.pas
Test/testt.pas    Test/testu.pas    Test/testv.pas    Test/testw.pas
Test/testx.pas    Test/testy.pas    Test/testz.pas    Test/sem1.pas
Test/sem2.pas Test/sem3.pas Test/sem4.pas Test/sem5.pas))

   (mapcar #'parser filer)

)

(defun doParse()

   (format t "Enter filename: ")
   (parser (read t))
)
```