LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Språkdokumentation

Författare

Gustav Melefors, gusme799@student.liu.se
Lukas Freyland, lukfr510@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023
Version 1.0

2023-04-10

Innehåll

1 Inledning

Det här projektet, konstruktion av datorspråk, är gjort under kursen *TDP019: Datorspråk* av två studenter som studerar Innovativ Programmering under våren 2023. Språket är modellerat efter C++ med liknande konstruktioner fast det har även en Dungeons & Dragons tema bakom sig.

2 Användarhandledning

Detta språk är uppbyggt utav olika konstruktioner, nedan följer en beskrivning utav samtliga konstruktioner och dess användning.

2.1 Main()

D++ avslutar varje program med en main funktion, som körs i början av varje körning. Den skrivs enligt följande:

int main() {

}

Inuti måsvingarna skrivs koden som ska köras, under kommande sektioner beskrivs vad som kan skrivas inuti dessa.

2.2 Grundläggande datatyper

I D++ är de grundläggande datatyper:

- **Integer**
- **Char**
- **Float**
- **Boolean**
- **String**

Integers är positiva eller negativa heltal, **Char** är enstaka karaktärer, till exempel a, b och c och skrivs ut med två apostrofer: 'a', 'b' och 'c', **Float** är decimal tal, **Boolean** är sanningsvärden **True** eller **False** och **String** är en eller fler karaktärer och skrivs med två citattecken; "hej".

2.3 Variabler

I D++ går det att tilldela värden till variabler. För att tilldela ett värde av en viss datatyp, säg **Float**, måste variabeln deklareras med den datatypen. Ett exempel på det är att spara undan värdet på pi (3.14) i variabeln x:

float x = 3.14

Här deklareras variabeln **x** som en **float** och tilldelas värdet 3.14 som är av typen **float**. En variabel kan endast bli tilldelad värden av samma datatyp. Det går till exempel inte att tilldela en **integer** till variabeln **x** efter att den har blivit deklarerad som en **float**.

2.4 Logik & Aritmetik

I D++ finns det logiska och aritmetiska uttryck. Ett logiskt uttrycks består av två uttryck och en symbol och skrivs ut på följande sätt: *rhs symbol lhs* (rhs = *right-hand-side*, lhs = *left-hand-side*). Ett logiskt uttryck ger ett **True-** eller **False-påstående** beroende på om uttrycket stämmer eller inte. De symboler som går att använda är:

- || : eller
- **&&** : och
- **!=** : Inte lika med
- == : Lika med
- <= : Mindre eller lika med
- >= : Större eller lika med
- < : Mindre än
- > : Större än

Några exempel på hur logiska uttryck andvänds och skrivs ut:

- True || False : True
- False || False : False
- True && False : False
- True && True : False
- True != False : True
- False == False : True
- 10 < 100 : True
- 10 > 100 : False

Aritmetiska uttryck ger antingen summan, kvoten, produkten, differensen eller resten av ett uttryck. De aritmetiska uttrycken är olika mattematiska uträkningar som likt de logiska uttrycken skrivs ut *rhs symbol lhs*. Symbolerna som används är:

- + : Addition
- **-** : Subtraktion
- ^ : Potens
- / : Division
- ***** : Multiplikation
- % : rest

Det går även att skapa nästlade uttryck där *rhs* är ett uttryck och *lhs* är ett annat uttryck. Det ser ut på liknande sätt:

(1 < 10) && (500 != 700)

Parenteser indikerar vad som är egna uttryck över det globala uttrycket. Det går också att göra ett logiskt uttryck med aritmetiska uttryck och logiska uttryck. Till exempel **2 + 3 == 5** ger **True**.

2.5 Print()

Print() är en del av språket och har en funktion att skriva ut valda saker i terminalen. Ett exempel där **print()** används för att skriva ut nummer 1.

int main() {

print(1)

}

Utskriften som kommer upp i terminalen är 1.

2.6 Komplexa datatyper

D++ har två typer av komplexa datatyper: en lista och en stack.

2.6.1 lista

En lista är en behållare som kan förvara x antal av en datatyp. En lista deklareras på följande sätt:

int[] tal = [4, 3, 2, 1]

int är vilken typ av datatyp listan ska förvara, i det här fallet **integers** och hakparenteserna visar att det är en lista som deklareras. **tal** är variabelnamnet på listan. Det är namnet som den här listan är refererad till. **[4, 3, 2, 1]** är listan vi vill tilldela variabeln **tal**. Listan representeras av två hakparenteser som innesluter de fyra siffrorna.

Addition samt subtraktion går att använda för listor i D++. Det går att föra ihop två separata listor in till en lista.

Lista har också funktionaliteten att ta fram ett specifikt element ur listan med hjälp av **index**. T listan i övre exemplet, för att ta fram det första värdet i listan skrivs det ut på följande sätt:

tal[1],

tal[1] är i det här faller **4**. Andra talet i listan skrivs då ut som **tal[2]** vilket är 3.

2.6.2 Stack

D++ inkluderar ett standardbilbotek med för-definierade klasser, det finns i nuläget en klass tillgänglig, stack. Stack har tre medlems funktioner: pop, push och peek. Pop tar bort översta elementet, push lägger ett element överst på stacken, och peek kollar på det översta elementet.

Stack kan användas endast med **integers** och supportar endast de till sin push metod.

2.7 Satser

D++ har olika satser som fungerar som styrstrukturer. En styrstruktur är en bit kod som påverkar ordningen i hur koden körs. De här styrstrukturerna representeras av **if-**, **else if-** och **else-**satser. En If-sats skrivs på följande sätt:

if (*Logiskt uttryck*)

{}

En **else if**- och **else**-sats används alltid i samband med en **if**-sats och ser ut såhär:

if (*logiskt uttryck nr. 1*)

{}

else if (*logiskt uttryck nr. 2*)

{}

else

{}

Hur de här satserna fungerar ihop är att om *logiskt uttryck nr. 1* är **falskt** kommer koden hoppa över den **if**-satsens kodblock och fortsätter ner till **else if**-satsen och kontrollerar om *logiskt uttryck nr. 2* är **sann**. Om det är **sann** körs det kod-blocket och programmet går ur satsen. Om uttrycket är **falskt** fortsätter koden ner till **else**-satsen och kör det kodblocket.

2.8 Loopar

Det finns två typer av loopar i D++; **while**-loopar och **for**-loopar. En **while**-loop upprepar kod tills ett bestämt krav har uppfyllts. En **while**-loop i D++ skrivs på följande sätt:

while (*logiskt uttryck*)

{

Kod som ska upprepas;

}

Här kommer koden i blocket att upprepas så länge det *logiska uttrycket* är **sant**. När det *logiska uttrycket* blir **falskt** bryts loopen.

For-loopar upprepar kod ett bestämt antal gånger och ser ut på följande sätt:

for (int i = 0; i = i + 1; i < 10)

{

Kod som ska upprepas;

}

Den här **for**-loopen upprepar koden i blocket 10 gånger. **For**-loopen fungerar så att en variabel **i** deklareras som en integer med värdet 0 (**int i = 0**). Sedan kommer **i**:s värde att öka med ett för varje upprepning (**i = i + 1**). Till sist så finns ett logiskt uttryck som beskriver vilket värde **i** ska bli för att upprepandet ska avbrytas(**i < 10**). På så sätt kan koden upprepas ett bestämt antal gånger.

2.9 Funktioner

I D++ kan funktioner definieras och måste definieras innan main funktionen. En funktion kan använda sig utav hur många argument som möjligt, och kan skrivas på följande sett:

int namnPåFunktionen() {}

Där efter kan den anropas genom att skriva **namnPåFunktionen()**. Intuti funktionen kan **return** användas för att funktionen ska ge tillbaka ett värde.

2.10 Klasser

I detta språk kan klasser skapas, dessa är behållare av både funktioner och variabler. Klasser följer en specifik kodstruktur där variabler deklareras först sedan konstruktör följt av funktioner. En klass skrivs på följande sett:

class NamnPåKlassen {

float x;

NamnPåKlassen() {};

void foo() {};

}

Klassen inledes med nyckelordet **class** följt av namnet på klassen och måste vara definerad innan main funktionen. Sedan kommer variabel deklarationen, här definieras namnet och typen på klassens variabler.

Efter att alla variabler är deklarerade kommer en konstruktör, som har samma namn som klassen, den här funktion körs alltid varje gång klassen skapas, det kan endast finnas en konstruktör i varje klass.

Efter att konstruktören kommer klassens tillhörande funktioner. De använder ett nyckelord **this** för att komma åt och använda de tillhörande funktioner och variabler. Det skrivs på följande sett:

this.[namn på funktion eller variabel]

För att skapa instanser av klassen skrivs det på följande sett:

KlassNamn instansensNamn = **new** KlassNamn();

Där klassnamnet är namnet på den definierade klassen och instansnamnet är namnet hos den nya klass instansen. Efter att en instans har skapats kan funktioner och variabler kommas åt med hjälp av en punkt mellan instansnamnet och variabel eller funktions namnet.

2.11 Illustrativt exempel

Nedan följer ett exmepel på hur kod kan skrivas i D++. Den inkluderar en metod med rekursion för att beräkna fakulteten hos ett tal.

```
class Car {
    int x;

    Car(int one) {
        this.x = one;
    };

    int fac(int val) {
        if (val == 1) {
            return 1;
        };
        return val * this.fib(val - 1);
    };

    int foo(int one) {
        print(this.x);
        print( one );
        this.x = one;
    };
    int bar(int zero, int two) {
        print(zero % two);
    };
    int foobar() {};
}


int main() {
    Car volvo = new Car( 20 );
    print(volvo.x);
    volvo.x = 5;
    print(volvo.x);
    volvo.foo(3);
    volvo.bar(5, 2);
    volvo.foobar();
    print(volvo.x);
    print(volvo.fac(5));
}
```

3 Systemdokumentation

Systemet är uppdelat i tre delar:

- RDparse
- Grammer
- Abstrakt syntax träd (AST)
- Standard bibliotek (STD)

Grammer där definitionen av grammatiken och den lexekaliska analysen finns. **RDparse** används för att läsa in det skrivna språket och identifiera alla rader och strängar samt hur de resulterande tokens ska matchas mot grammatiken. **AST** är funktionaliteten av språket vilket gör det möjligt att göra olika saker i D++, här finns alla noder som gör upp det AST definerade. **STD** är standard biblioteket i D++ där det finns fördefinierade klasser av till exempel en komplex datatyp.

3.1 Grammatiken

Nedan följer en detaljerad beskriving av språkets grammtik och dess regler i Extended Backus Backus-Naur Form.

3.1.1 Main

⟨*main*⟩ ::= '`int`' '`main`' '`(`' '`)`' ⟨*block*⟩
| ⟨*class*⟩ ⟨*main*⟩
| ⟨*function*⟩ ⟨*main*⟩

3.1.2 Block

⟨*block*⟩ ::= '`{`' ⟨*statements*⟩ '`}`'`;`'
| '`{`' '`}`'

⟨*statements*⟩ ::= ⟨*statement*⟩ '`;`' ⟨*statements*⟩
| ⟨*statement*⟩ '`;`'

⟨*statement*⟩ ::= ⟨*list*⟩
| ⟨*findelement*⟩
| ⟨*varset*⟩
| ⟨*if*⟩
| ⟨*boolean*⟩
| ⟨*for*⟩
| ⟨*while*⟩
| ⟨*return*⟩
| ⟨*print*⟩

3.1.3 Inbyggda funktioner

⟨*print*⟩ ::= '`print`' '`(`' ⟨*boolean*⟩ '`)`'

⟨*return*⟩ ::= '`return`' ⟨*boolean*⟩

3.1.4 Klasser

⟨*class*⟩ ::= '`class`' ⟨*classidentifier*⟩ '`{`' ⟨*classvarblock*⟩ ⟨*classconstructor*⟩ ⟨*classfuncblock*⟩ '`}`'
| '`class`' ⟨*classidentifier*⟩ '`{`' ⟨*classvarblock*⟩ ⟨*classconstructor*⟩ '`}`'
| '`class`' ⟨*classidentifier*⟩ '`{`' ⟨*classconstructor*⟩ ⟨*classfuncblock*⟩ '`}`'
| '`class`' ⟨*classidentifier*⟩ '`{`' '`}`'

⟨*classconstructor*⟩ ::= ⟨*classidentifier*⟩ ⟨*_params*⟩ ⟨*block*⟩ '`;`'

⟨*classvarblock*⟩ ::= ⟨*primitive*⟩ ⟨*identifier*⟩ '`;`' ⟨*classvarblock*⟩
| ⟨*primtive*⟩ ⟨*identifier*⟩ '`;`'

⟨*classfuncblock*⟩ ::= ⟨*primitive*⟩ ⟨*identifier*⟩ ⟨*_params*⟩ ⟨*block*⟩ '`;`' ⟨*classfuncblock*⟩
| ⟨*primitive*⟩ ⟨*identifier*⟩ ⟨*_params*⟩ ⟨*block*⟩ '`;`'

⟨*_params*⟩ ::= '`(`'`)`'
| '`(`' ⟨*params*⟩ '`)`'

⟨*classinit*⟩ ::= '`new`' ⟨*classIdentifier*⟩ ⟨*callparams*⟩

⟨*classIdentifier*⟩ ::= /\A[A-Z]\w*/

⟨*classmethod*⟩ ::= ⟨*identifier*⟩ '`.`' ⟨*identifier*⟩ '`(`' '`)`'
| ⟨*identifier*⟩ '`.`' ⟨*identifier*⟩ '`(`' ⟨*callparams*⟩ '`)`'

3.1.5 Funktions anropp och deklaration

⟨*function*⟩ ::= ⟨*primitive*⟩ ⟨*identifier*⟩ '`(`' ⟨*params*⟩ '`)`' ⟨*block*⟩
| ⟨*primitive*⟩ ⟨*name*⟩ '`(`' '`)`' ⟨*block*⟩

⟨*call*⟩ ::= ⟨*identifier*⟩ '`(`'⟨*callparams*⟩´)'
| ⟨*identifier*⟩ '`(`' '`)`'

⟨*params*⟩ ::= ⟨*params*⟩ '`,`' ⟨*param*⟩
| ⟨*param*⟩

⟨*param*⟩ ::= ⟨*primitive*⟩ ⟨*name*⟩

⟨*callparams*⟩ ::= ⟨*callparams*⟩ '`,`' ⟨*boolean*⟩
| ⟨*boolean*⟩

3.1.6 While loop och for-loop

⟨*while*⟩ ::= '`while`' '`(`' ⟨*boolean*⟩ '`)`' ⟨*block*⟩

⟨*for*⟩ ::= '`for`' '`(`' ⟨*varset*⟩ '`;`' ⟨*varset*⟩ '`;`' ⟨*boolean*⟩ ´)' ⟨*block*⟩

3.1.7 Komplexa datatyper

⟨*list*⟩ ::= ⟨*primitive*⟩ '`[`' '`]`' ⟨*identifier*⟩
| ⟨*primitive*⟩ '`[`' '`]`' ⟨*name*⟩ '`=`' ⟨*primlist*⟩
| ⟨*primitive*⟩ '`[`' '`]`' ⟨*name*⟩ '`=`' '`[`' '`]`'

⟨*primlist*⟩ ::= '`[`' ⟨*members*⟩ '`]`'

⟨*members*⟩ ::= ⟨*members*⟩ '`,`' ⟨*member*⟩
| ⟨*member*⟩

⟨*member*⟩ ::= ⟨*var*⟩

⟨*findelement*⟩ ::= ⟨*identifier*⟩ '`[`' :int '`]`'

3.1.8 If satser

⟨*if*⟩ ::= '`if`' '`(`' ⟨*boolean*⟩ '`)`' ⟨*block*⟩ ⟨*else*⟩
| '`if`' '`(`' ⟨*boolean*⟩'`)`' ⟨*block*⟩

⟨*else*⟩ ::= '`else`' ⟨*block*⟩
| '`else`' '`if`' '`(`' ⟨*boolean*⟩ '`)`' ⟨*block*⟩ ⟨*else*⟩
| '`else`' '`if`' '`(`' ⟨*boolean*⟩ '`)`' ⟨*block*⟩
| '`else`' ⟨*block*⟩

3.1.9 Logik och aritmetik

⟨*boolean*⟩ ::= ⟨*boolean*⟩ '`||`' ⟨*and*⟩
| ⟨*and*⟩

⟨*and*⟩ ::= ⟨*and*⟩ '`&&`' ⟨*relation*⟩
| ⟨*relation*⟩

⟨*relation*⟩ ::= ⟨*relation*⟩ '!=' ⟨*addition*⟩
| ⟨*relation*⟩ '`==`' ⟨*addition*⟩
| ⟨*relation*⟩ '`<`' ⟨*addition*⟩
| ⟨*relation*⟩ '`>`' ⟨*addition*⟩
| ⟨*relation*⟩ '`<=`' ⟨*addition*⟩
| ⟨*relation*⟩ '`<=`' ⟨*addition*⟩
| ⟨*addition*⟩

⟨*addition*⟩ ::= ⟨*addition*⟩ '`+`' ⟨*multi*⟩
| ⟨*addition*⟩ '`-`' ⟨*multi*⟩
| ⟨*multi*⟩

⟨*multi*⟩ ::= ⟨*term*⟩ '`^`' ⟨*multi*⟩
| ⟨*multi*⟩ '`/`' ⟨*term*⟩
| ⟨*multi*⟩ '`%`' ⟨*term*⟩
| ⟨*term*⟩

⟨*term*⟩ ::= '`(`' ⟨*boolean*⟩ '`)`'
| '`True`'
| '`False`'
| ⟨*var*⟩

⟨*var*⟩ ::= ⟨*integer*⟩
| ⟨*classmethod*⟩
| ⟨*call*⟩
| ⟨*float*⟩
| ⟨*int*⟩
| ⟨*char*⟩
| ⟨*string*⟩
| ⟨*primlist*⟩
| ⟨*varget*⟩

3.1.10 Variable

⟨*float*⟩ ::= ⟨*int*⟩ '.' ⟨*int*⟩
⟨*int*⟩ ::= /\d+/
⟨*char*⟩ ::= '"'/\w/'"'
⟨*identifier*⟩ ::= /\A[a-z]\w*/
⟨*varget*⟩ ::= ⟨*findelement*⟩
| ⟨*identifier*⟩ '.' ⟨*identifier*⟩
| ⟨*identifier*⟩
⟨*varset*⟩ ::= ⟨*primitive*⟩ ⟨*identifier*⟩ '=' ⟨*varset2*⟩
| ⟨*identifier*⟩ '.' ⟨*identifier*⟩ '=' ⟨*varset2*⟩
| ⟨*identifier*⟩ ´=' ⟨*varset2*⟩
⟨*varset2*⟩ ::= ⟨*classinit*⟩
| ⟨*boolean*⟩

3.1.11 Primitiver

⟨*primitive*⟩ ::= '`char`'
| '`int`'
| '`float`'
| '`bool`'
| '`string`'
| '`void`'
| ⟨*classIdentifier*⟩

3.2 Kodstandard

Kodstandarden som följts under projektets gång:

- Funktioner som tar variabler skrivs med parenteser.
- Namn på variabler är tydliga och beskriver vilket värde variabeln är tilldelad.
- Kod indenteras under funktioner, satser samt loopar.

3.3 Packetering & användandet av språket

För att använda sig utav språket, görs enklast med att skapa ett nytt DnD parser objekt i ruby och parsa filen med hjälp av **parse** funktionen, som låter en skriva in ett filnamn som programmet ska parsa och köra.

3.4 Parsning

Parsning behandlar hur ett formellt språk bryts ner i dess beståndsdelar för att ett uttryck som är skrivet i språket ska tolkas.

3.4.1 Lexikalisk analys

Lexikalisk analys är konverteringen av en sträng av tecken till en lista av symboler (tokens på engelska).

3.4.2 Parser

I parsern förbrukas alla symboler som skapades av den lexikaliska analysen. Parsen är en rekursiv nertraversering parser vilket betyder att det är en *Top-Down Parser* och det i sin tur betyder att parsen bygger trädet från överst till lägst och analyserar rekursivt på samma sätt.

3.4.3 Abstrakt syntax Träd

Abstrakta syntax trädet är representerat av klasser som alla gemensamt ärver ifrån **Node** klassen. Alla nod klasser innehåller en gemensam funktion, evaluate, som är körningen på programmet.

Node klassen innehåller en statisk medlems variabel **stackframe** samt tillhörande hjälp funktioner. Stackframe är en hash som innehåller alla variabler, funktioner och klass definitioner. Stackframe innehåller all information som är relevant under runtime.

4 Erfarenheter och reflektion

Vi planerade att implementera polymorfi men på grund av att klasser tog längre tid att implementera än planerat blev detta uppskjutet och till slut fick vi lägga undan det pga tidsbristen. Det var även planerat att göra string till en komplex datatyp, men det beslutades senare att det skulle vara enklare och mer logiskt att ha string vara en simpel datatyp.

Vi hade från början tanken att lista skulle vara den ordentliga behållaren i D++ med tillhörande operationer. Dock så märkte vi att det blev krångligt då vi inte ville att operationerna skulle vara inbyggda i språket. Vi löste det här genom att skapa ett standard bibliotek där vi skapade klassen stack som en till container. Stack har flera medlemsfunktioner som fungerar som tillhörande operatiorer.

En sak vi har haft svårt med mot slutet av projektet är en bug för stack. Push-funktionen för att lägga till element i en stack blir konstig när man lägger till fler än ett tal. Om två olika tal läggs kommer stacken att innehålla två element av det första talet som skickades in. Vi har felsökt koden samt stackframen för att försöka hitta vad som orsakar bugget men koden såg ut att fungera exakt som den ska för att fungera. Det här beteendet är något vi inte förstår oss på och har haft ont om tid att genomsöka. För att lösa det skulle vi behövt använda oss av en mer robust IDE.

Tanken var från början att språket skulle ha en Dungeon generator samt Dungeons and Dragons tema men pågrund av tidsbrist så hade vi bara tid med att implementera större delen av det generella språket.

Över lag har vi följt implementationsplanen och lyckats hålla oss i fas större delen av projektets gång.

Vi har lärt oss mer om hur ett språk byggsupp och kan se hur det här kan vara användbart i framtiden. Det har tagit bort en nivå av mystik från hur programmeringsspråk fungerar och gett mer insikt i vad som bakom kulisserna på språken.