

TDP019 Projekt: Datorspråk

Dungeon++

Författare

Gustav Melefors, gusme799@student.liu.se

Lukas Freyland, lukfr510@student.liu.se

Innehåll

1	Inledning	2
2	Användarhandledning	2
2.1	Main()	2
2.2	Grundläggande datatyper	2
2.3	Variabler	2
2.4	Logik och aritmetik	3
2.5	Print()	4
2.6	Komplexa datatyper	4
2.6.1	Lista	4
2.6.2	Stack	4
2.7	Satser	5
2.8	Loopar	5
2.9	Funktioner	6
2.10	Klasser	6
2.11	Illustrativt exempel	7
3	Systemdokumentation	8
3.1	Grammatiken	8
3.1.1	Main	8
3.1.2	Block	8
3.1.3	Inbyggda funktioner	8
3.1.4	Klasser	9
3.1.5	Funktionsanrop och deklaration	9
3.1.6	While-loop och for-loop	9
3.1.7	Komplexa datatyper	9
3.1.8	If-satser	10
3.1.9	Logik och aritmetik	10
3.1.10	Variable	11
3.1.11	Primitiver	11
3.2	Kodstandard	11
3.3	Packetering och användandet av språket	11
3.4	Parsning	11
3.4.1	Lexikalisk analys	11
3.4.2	Parser	12
3.4.3	Abstrakt syntaxträd	12
4	Erfarenheter och reflektion	12

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 ett Dungeons & Dragons-tema bakom sig. Målgruppen till D++ är Dungeons & Dragons intresserade utvecklare.

2 Användarhandledning

Detta språk är uppbyggt av 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 skrivs a, b och c ut med två apostrofer: 'a', 'b' och 'c'. **Float** är decimaltal. **Boolean** är sanningsvärdena **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 variabler värden. För att tilldela ett värde av en viss datatyp, t.ex. **Float**, måste variabeln deklarerar med den datatypen. Ett exempel på det är att spara undan värdet på pi (3.14) i variabeln x:

```
float x = 3.14
```

Här deklarerar 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 och aritmetik

I D++ finns det logiska och aritmetiska uttryck. Ett logiskt uttryck 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 används 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 matematiska 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 följande 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 deklarerar på följande sätt:

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

int är vilken slags datatyp listan ska förvara. I det här fallet visar **integers** och hakparenteserna att det är en lista som deklarerar. **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 till en lista.

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

```
tal[1],
```

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

2.6.2 Stack

D++ inkluderar ett standardbibliotek med för-definierade klasser. Det finns i nuläget en klass tillgänglig, **stack**. **Stack** har två medlemsfunktioner: **pop** och **push**. **Pop** tar bort översta elementet medan **push** lägger ett element överst på stacken.

Stack kan användas endast med **integers**.

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 A)
```

```
{
```

```
else if (logiskt uttryck B)
```

```
{
```

```
else
```

```
{
```

Dessa fungerar tillsammans genom att när *uttryck A* värderas, om den är **sann** kommer kod blocket exekveras men varken *uttryck B* eller *else* kommer utvärderas. *Uttryck B* utvärderas endast om uttryck A är **falskt**. Sista **else** satsen utvärderas endast om alla förgående uttryck är **falska**.

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 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 finns ett logiskt uttryck som beskriver vilket värde **i** ska bli för att upprepadet 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 av hur många argument som möjligt, och kan skrivas på följande sett:

```
int namnPåFunktionen() {}
```

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

2.10 Klasser

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

```
class NamnPåKlassen {  
    float x;  
    NamnPåKlassen() {};  
    void foo() {};  
}
```

Klassen inleds med nyckelordet **class** följt av namnet på klassen och måste vara definerad innan main-funktionen. Sedan kommer variabel-deklarationen och 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 funktionen körs alltid varje gång klassen skapas och det kan endast finnas en konstruktör i varje klass.

Efter att konstruktören är klar kan klassens tillhörande funktioner skapas. De använder ett nyckelord, **this**, för att komma åt och använda de tillhörande funktionerna och variablerna. Det skrivs på följande sätt:

```
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 klassinstansen. Efter att en instans har skapats går det att komma åt funktioner och variabler med hjälp av en punkt mellan instansnamnet och variabel eller funktionsnamnet.

2.11 Illustrativt exempel

Nedan följer ett exempel 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 syntaxträd (AST)
- Standard bibliotek (STD)

Grammer definieras av grammatiken och där finns lexikaliska analysen. **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 som definierade. **STD** är standardbiblioteket i D++ där det finns fördefinierade klasser av till exempel en komplex datatyp.

3.1 Grammatiken

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

3.1.1 Main

```
 $\langle main \rangle ::= \text{'int' 'main' '(' ')} \langle block \rangle$   
          |  $\langle class \rangle \langle main \rangle$   
          |  $\langle function \rangle \langle main \rangle$ 
```

3.1.2 Block

```
 $\langle block \rangle ::= \text{'{' } \langle statements \rangle \text{'}' ';'}$   
          |  $\text{'{' '}'}$   
 $\langle statements \rangle ::= \langle statement \rangle \text{';' } \langle statements \rangle$   
              |  $\langle statement \rangle \text{';'}$   
 $\langle statement \rangle ::= \langle list \rangle$   
              |  $\langle varset \rangle$   
              |  $\langle if \rangle$   
              |  $\langle boolean \rangle$   
              |  $\langle for \rangle$   
              |  $\langle while \rangle$   
              |  $\langle return \rangle$   
              |  $\langle print \rangle$ 
```

3.1.3 Inbyggda funktioner

```
 $\langle print \rangle ::= \text{'print' '(' } \langle boolean \rangle \text{'')}$   
 $\langle return \rangle ::= \text{'return' } \langle boolean \rangle$ 
```

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⟩
                  | ⟨primitive⟩ ⟨identifier⟩ ';'
                  | ⟨primitive⟩ '[' ']' ⟨identifier⟩ ';' ⟨classvarblock⟩
                  | ⟨primitive⟩ '[' ']' ⟨identifier⟩ ';'

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

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

⟨classinit⟩ ::= 'new' ⟨classidentifier⟩ '(' ⟨callparams⟩ ')'
             | 'new' ⟨classidentifier⟩ '(' ']'

⟨classidentifier⟩ ::= /\A[A-Z]\w*/

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

```

3.1.5 Funktionsanrop och deklaration

```

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

⟨call⟩      ::= ⟨identifier⟩ '(' ⟨callparams⟩ ')'
             | ⟨identifier⟩ '(' ']'

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

⟨param⟩     ::= ⟨primitive⟩ ⟨identifier⟩

⟨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

$\langle if \rangle ::= 'if' '(' \langle boolean \rangle ')' \langle block \rangle \langle else \rangle$
 $\quad \quad \quad | 'if' '(' \langle boolean \rangle ')' \langle block \rangle$
 $\langle else \rangle ::= 'else' \langle block \rangle$
 $\quad \quad \quad | 'else' 'if' '(' \langle boolean \rangle ')' \langle block \rangle \langle else \rangle$
 $\quad \quad \quad | 'else' 'if' '(' \langle boolean \rangle ')' \langle block \rangle$
 $\quad \quad \quad | 'else' \langle block \rangle$

3.1.9 Logik och aritmetik

$\langle boolean \rangle ::= \langle boolean \rangle '||' \langle and \rangle$
 $\quad \quad \quad | \langle and \rangle$
 $\langle and \rangle ::= \langle and \rangle ' \& \& ' \langle relation \rangle$
 $\quad \quad \quad | \langle relation \rangle$
 $\langle relation \rangle ::= \langle relation \rangle '!= ' \langle addition \rangle$
 $\quad \quad \quad | \langle relation \rangle '== ' \langle addition \rangle$
 $\quad \quad \quad | \langle relation \rangle '< ' \langle addition \rangle$
 $\quad \quad \quad | \langle relation \rangle '> ' \langle addition \rangle$
 $\quad \quad \quad | \langle relation \rangle '<= ' \langle addition \rangle$
 $\quad \quad \quad | \langle relation \rangle '<= ' \langle addition \rangle$
 $\quad \quad \quad | \langle addition \rangle$
 $\langle addition \rangle ::= \langle addition \rangle '+' \langle multi \rangle$
 $\quad \quad \quad | \langle addition \rangle '-' \langle multi \rangle$
 $\quad \quad \quad | \langle multi \rangle$
 $\langle multi \rangle ::= \langle term \rangle '^' \langle multi \rangle$
 $\quad \quad \quad | \langle multi \rangle '/' \langle term \rangle$
 $\quad \quad \quad | \langle multi \rangle '%' \langle term \rangle$
 $\quad \quad \quad | \langle term \rangle$
 $\langle term \rangle ::= '(' \langle boolean \rangle ')'$
 $\quad \quad \quad | 'True'$
 $\quad \quad \quad | 'False'$
 $\quad \quad \quad | \langle var \rangle$
 $\langle var \rangle ::= \langle classmethod \rangle$
 $\quad \quad \quad | \langle call \rangle$
 $\quad \quad \quad | \langle float \rangle$
 $\quad \quad \quad | \langle int \rangle$
 $\quad \quad \quad | \langle char \rangle$
 $\quad \quad \quad | \langle string \rangle$
 $\quad \quad \quad | \langle primlist \rangle$
 $\quad \quad \quad | \langle varget \rangle$

3.1.10 Variable

```

<string>    ::= ‘”’ /.*/ ‘”’
<float>     ::= <int> ‘.’ <int>
<int>       ::= /\d+/
<char>      ::= ‘”’/\w/‘”’
<identifier> ::= /\A[a-z]\w*/
<varget>    ::= <findelement>
              | <findelement>
              | <identifier> ‘.’ <identifier> ‘[’ <var> ‘]’
              | <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’
              | <primitive> ‘[’ ‘]’
              | /\A[A-Z]\w*/

```

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 som variabeln är tilldelad.
- Kod indenteras under funktioner, satser samt loopar.

3.3 Packetering och användandet av språket

För att använda sig av språket, görs enklast med att skapa ett nytt DnD parser-objekt i Ruby och parsar filen med hjälp av **parse**-funktionen, som låter en skriva in ett filnamn som programmet ska parsar 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 syntaxträd

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 medlemsvariabel **stackframe** samt tillhörande hjälpfunktioner. Stackframe är en hash som innehåller alla variabler, funktioner och klassdefinitioner. 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 om string är en simpel datatyp.

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

En sak vi har haft svårt med mot slutet av projektet är en bugg 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 stackramen för att försöka hitta vad som orsakar buggen men koden såg ut att fungera exakt som den ska för att fungera. Det här beteendet är något vi inte förstode oss på och har haft ont om tid att genomsöka. Efter två timmar av felsökande lyckades vi hitta problemet och det uppstod i en av våra noder som används då en typlös och namnlös lista deklarerar. Problemet var att i noden så uppdaterades aldrig värdena i listan. Noden använde alltså värdet som den fick första gången koden kördes alla gånger koden kördes. Det var en enkel lösning och det ända vi gjorde var att tilldela en variabel värdet som noden får i början av evalueringen och använde den variabeln istället för att direkt använda den klassdeklarerade variabeln.

Tanken var från början att språket skulle ha en Dungeon-generator samt Dungeons and Dragons-tema men på grund 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 byggs upp 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 sker bakom kulisserna för olika språk.