

Gameparser

Linköpings universitet
Innovativ Programmering
TDP019 - Projekt: Datorspråk
Version 1



Gameparser

Vincent Ahlström, vinah331@student.liu.se
Hadi Ansari, hadan236@student.liu.se

	1
1. Introduktion	2
Syfte	2
Målgrupp	2
2. Användarhandledning	3
Installation	3
Konstruktioner	3
Datatyper	4
3. Språkstruktur	5
Kommentar	5
Variabel	5
Operatorer	8
Sträng-operatorer	10
List-operationer	11
In-och utmatning	13
Villkor	14
Repetitionssatser	15
Break	16
Funktioner	17
Rekursion	18
Enkla klasser	19
event	19
prop	21
Scope	23
Andra fördefinierade funktioner	24
4. Systemdokumentation	25
Lexer	25
Parser	25
Abstrakt syntaxträd	25
Kodstandard	25
BNF	26
5. Reflektion	30

1. Introduktion

I den här rapporten skriven som en del av kursen TDP019 Projekt: Datorspråk, kommer programmeringsspråket Gameparser dokumenteras. Projektet Datorspråk är det tredje projektet i programmet innovativ programmering och det handlar om att utveckla ett eget datorspråk. Projektet utvecklades av en grupp av två IP-studenter under andra perioden av vårterminen 2021.

Språket är designat för (men ej begränsat till) att skapa textbaserade spel. Det har sin grund i Ruby där det via `rdparse.rb` översätter skriven kod till Ruby för körning. Syntaxen är inspirerad av både Ruby och Python vilket gör språket användarvänligt. Gameparser ämnar att erbjuda användaren alla verktyg som krävs för att designa en interaktiv och levande spelvärld i ett textbaserat format.

Syfte

Projektet ämnade ge deltagarna djupare förståelse för hur programmeringsspråk exekveras samt hur olika designval kan påverka dessa.

Målgrupp

Gameparser är skapat för individer som vill skapa textbaserade spel och har programmerat lite innan. För den något mer erfarne programmeraren möjliggör Gameparser skapandet av egna funktioner och klasser.

2. Användarhandledning

Här beskrivs hur språket kan installeras och användas samt information om vilka konstruktioner och datatyper språket har stöd för.

Installation

För att kunna komma igång med systemet två verktyg behöver installeras:

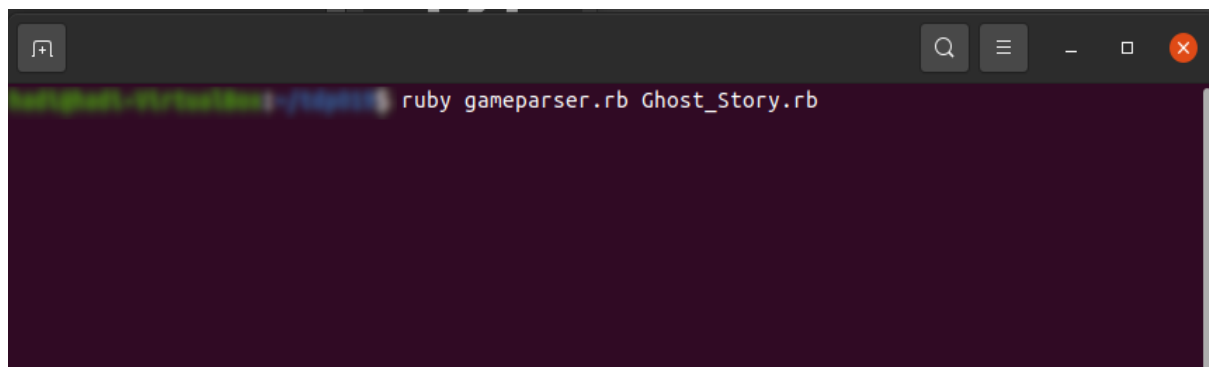
1. Installera Ruby (version 2.7 rekommenderas)
2. Installera *fim* med *sudo apt-get install fim*.

Gameparser kan köras interaktivt genom att skriva *ruby gameparser.rb* i kommandotolken (se figur 1). Kod (spel) skrivs med fördel i separata dokument som sedan körs via *ruby gameparser.rb filväg*. Figur 2 visar hur ett spel i ett separat dokument kan köras med Gameparser.

A terminal window with a dark background. The prompt is '\$ ruby gameparser.rb'. The output shows '[gameParser]: 1 + 1' followed by '=> 2' and then '[gameParser]: ' with a cursor. The window has standard Linux window controls (minimize, maximize, close) and a search icon in the title bar.

```
$ ruby gameparser.rb
[gameParser]: 1 + 1
=> 2
[gameParser]: 
```

Figur 1: Att köra Gameparser i interaktivt-läge

A terminal window with a dark background. The prompt is 'ruby gameparser.rb Ghost_Story.rb'. The window has standard Linux window controls and a search icon in the title bar.

```
ruby gameparser.rb Ghost_Story.rb
```

Figur 2: Att köra ett spel med Gameparser

Konstruktioner

Språket har stöd för villkorssatser, repetitionssatser, funktioner och rekursion, enkla klasser och verktyg för att arbeta med listor samt några fördefinierade funktioner.

Med hjälp av en kombination av dessa verktyg har programmeraren möjlighet att designa ett unikt spel.

Datatyper

Gameparser har stöd för följande datatyper:

- **Integer:** som kan vara ett heltal (både positiva och negativa)
- **Sträng:** som kan vara en sträng av vilka karaktärer som helst (inklusive svenska bokstäver o.s.v)
- **Bool:** som kan antingen vara sann eller falsk
- **Lista:** som är en samling av olika datatyper inom hakparenteser
- **Range:** som är en spektrum av heltal med ett specificerat start/slut-värde.

Notera att en range kan båda inkludera och exkludera slutvärdet (se figur 3). I språket kan även en range som har ett större startvärde än slutvärdet skapas.

```
[gameParser]: my_range1 = (1..5)    # Inklusive slutvärdet  
=> [1, 2, 3, 4, 5]  
[gameParser]: my_range2 = (1...5)  # Exklusiv slutvärdet  
=> [1, 2, 3, 4]
```

Figur 3: Exempel på hur en range kan båda inkludera och exkludera slutvärdet

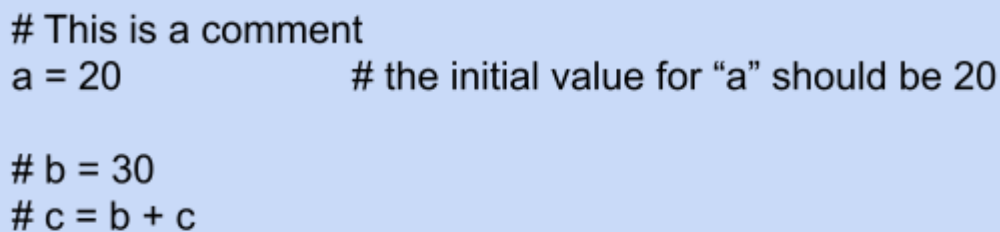
I avsnittet “Variabel” kommer fler exempel på hur dessa datatyper kan användas för att kunna deklarera olika typer av variabler.

3. Språkstruktur

Gameparser som språk är väldigt fritt. Ett helt spel skulle kunna skrivas på en rad om så önskas. Det finns därmed inga krav på indentering. Däremot har verktyg så som funktioner och klasser krav på hur de ska se ut. Block {} används för att köra kod medan parenteser markerar parametrar och prioriteringsordningar.

Kommentar

Gameparser tillåter kommentarer i koden för att programmeraren ska kunna förtydliga sin kod. Kommentarer representeras med ett “#” tecken framför kommentaren (se figur 4).



```
# This is a comment  
a = 20           # the initial value for "a" should be 20  
  
# b = 30  
# c = b + c
```

Figur 4: Exempel på hur koden kan kommenteras

Variabel

Deklaration av en variabel är väldigt enkelt. Ett variabelnamn (identifierare) inleds med en bokstav och namnet får inte vara ett reserverad nyckelord i språket. I figur 5 visas deklaration av olika typer av variabler.

```
[gameParser]: number = 100
=> 100
[gameParser]: color = "Red"
=> Red
[gameParser]: alive = true
=> true
[gameParser]: inventory = ["rusty sword", "magic boots"]
=> ["rusty sword", "magic boots"]
```

Figur 5: Exempel på deklaration av olika variabeltyper

Språket har ett dynamiskt typsystem vilket innebär att det inte finns strikta regler för att definiera variabler när det gäller typ. Det gör så att en typkontroll sker under exekvering. Därmed är det möjligt att skapa variabler av olika typer och ändra dem genom att tilldela ett annat värde av en annan typ om så önskas (se figur 6).

```
[gameParser]: var = 1
=> 1
[gameParser]: var = false
=> false
[gameParser]: var = "Hello"
=> Hello
[gameParser]: var = [1, true, "Hello"]
=> [1, true, "Hello"]
[gameParser]: var = (1..5)
=> [1, 2, 3, 4, 5]
```

Figur 6: Dynamisk typsystem i Gameparser

Det är även möjligt att göra en variabel-tilldelning med hjälp av variabler som redan har definierats (se figur 7).

```

[gameParser]: a = 1
=> 1
[gameParser]: b = a
=> 1
[gameParser]: c = [a, b]
=> [1, 1]
[gameParser]: d = [1, a, "Hello", b]
=> [1, 1, "Hello", 1]
[gameParser]: a = [a]
=> [1]

```

Figur 7: Deklaration av nya variabler med existerande variabler

Språket har stöd för en form av scope-hantering så att variabler i en funktion till exempel kan nås inom den ramen som är rimlig för dem. Samma regler gäller för andra konstruktioner så som “event” och “prop”. De har sina egna lokala variabler som inte kan nås utanför den rimliga ramen. Scope-hantering förklaras under rubriken “Scope” längre ned i dokumentationen.

I språket kan programmeraren skapa variabler i ett globalt scope som kan kallas var som helst i koden. Globala variabler definieras med hjälp av ett “\$” tecken som förekommer framför variabel-namnet (se figur 8).

```

[gameParser]: $player_name = "Kratos"
=> Kratos
[gameParser]: $player_name
=> Kratos

```

Figur 8: Deklaration av globala variabler

Alla regler som gäller vanliga variabler gäller även globala variabler. Den enda skillnaden är att globala variabler är tillgängliga överallt i koden och inte är beroende på scope (se figur 9).


```
[gameParser]: $a = 1
=> 1
[gameParser]: $b = $a
=> 1
[gameParser]: $c = [$a, $b]
=> [1, 1]
[gameParser]: $d = [1, $a, "Hello", $b]
=> [1, 1, "Hello", 1]
[gameParser]: $a = [$a]
=> [1]
```

Figur 9: Deklaration av nya globala variabler med existerande globala variabler

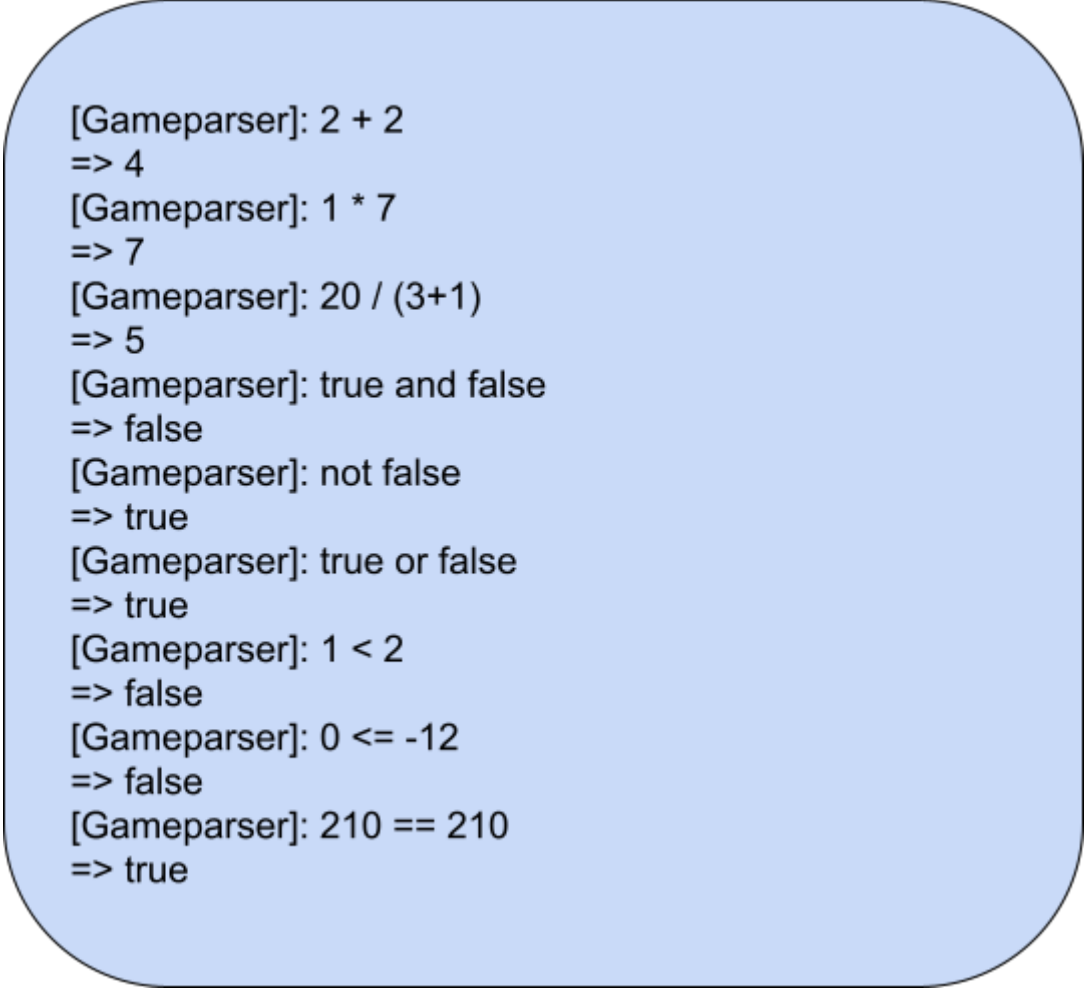
Det går också att tilldela en vanlig variabel till en global variabel eller tvärtom (se figur 10).

```
[gameParser]: a = 1
=> 1
[gameParser]: $a = a
=> 1
[gameParser]: $b = 2
=> 2
[gameParser]: b = $b
=> 2
```

Figur 10: Deklaration av nya globala/icke-globala variabler med existerande globala/icke-globala variabler

Operatorer

Gameparser inkluderar alla standardoperationer inom aritmetik, logik och jämförelser. Dessa följer de etablerade matematiska prioriteringsreglerna och logik tolkas i vänsterled. Utöver detta kan parenteser användas för ytterligare precision. Figur 11 visar några exempel på enklare aritmetiska och logiska uttryck. Se figur 12 för mer avancerade exempel som är tillåtna i Gameparser.



```
[Gameparser]: 2 + 2  
=> 4  
[Gameparser]: 1 * 7  
=> 7  
[Gameparser]: 20 / (3+1)  
=> 5  
[Gameparser]: true and false  
=> false  
[Gameparser]: not false  
=> true  
[Gameparser]: true or false  
=> true  
[Gameparser]: 1 < 2  
=> false  
[Gameparser]: 0 <= -12  
=> false  
[Gameparser]: 210 == 210  
=> true
```

Figur 11: Exempel på enklare aritmetiska och logiska uttryck i Gameparser

```

[gameParser]: (22/2) + (2 * 5) - (+20)
=> 1
[gameParser]: 2 / 2 + 10 * 2
=> 21
[gameParser]: --(+2) +-1 / (-++2)
=> -2
[gameParser]: --(++3)
=> 3
[gameParser]: (12 > 100) and (100 < 2)
=> false
[gameParser]: not ((12 > 10) or (100 < 2))
=> false
[gameParser]: 2 == 3 or 100 >= 200
=> false
[gameParser]: (true and false) or ( (false or true) and
true)
=> true
[gameParser]: false and true or true and false
=> false
[gameParser]: false and true or true
=> true

```

Figur 12: Mer avancerade aritmetiska och logiska uttryck

Sträng-operatorer

I språket finns ett antal operationer som kan utföras över strängar.

Utöver vanliga operatorer som finns är det också möjligt att addera strängar. För att ha ett citattecken i en sträng ska ett “\” förekomma framför citattecknet. Det gäller samma regler för “\n” som representerar ny rad och “\t” som representerar ett tab. **str()** kan användas för att konvertera andra datatyper till sträng. Figur 13 visar hur dessa operationer kan användas i Gameparser.

```

[gameParser]: "\"hello\""
=> "hello"
[gameParser]: "hello\n\n"
=> hello

[gameParser]: "hello\tworld!"
=> hello    world!
[gameParser]: "this" + " is " + " a " + " test"
=> this is  a test
[gameParser]: name = "Kratos"
=> Kratos
[gameParser]: greeting = "Hello "
=> Hello
[gameParser]: greeting + name
=> Hello Kratos
[gameParser]: str(1)
=> 1                # är strängen "1"
[gameParser]: str(true)
=> true             # är strängen "true"

```

Figur 13: Exempel på sträng-operatorer

List-operationer

Gameparser har stöd för specifika operationer för att göra arbetet med listor smidigare. De grundläggande operationer som är tillgängliga är att läsa ett element i listan och skriva över ett element. Språket är noll-indexerat vilket innebär det första elementet i listan har index noll (se figur 14).

```
[gameParser]: my_list = ["a", "b", "c"]  
=> ["a", "b", "c"]  
[gameParser]: my_list[0]  
=> a  
[gameParser]: my_list[1] = 2  
=> 2  
[gameParser]: my_list  
=> ["a", 2, "c"]
```

Figur 14: Tillgång till elementen i en lista

Utöver dessa finns även följande operationer i språket:

- **append:** Lägger till en element i slutet av listan (kan båda skrivas som **my_list.append(20)** och **my_list << 20**)
- **remove:** Tar bort ett element med det angivna indexet. Om index inte är angivet tas bort sista elementet i listan
- **insert:** Lägger till ett element på det angivna indexet med formatet **my_list.insert(element, index)**
- **len:** Returnerar en heltal som motsvarar antal element i listan.

Figur 15 visar exempel på alla list-operatorer som Gameparser har stöd för.

```
[gameParser]: my_list = [1,2,3]
=> [1, 2, 3]
[gameParser]: my_list.append(4)
=> [1, 2, 3, 4]
[gameParser]: my_list << 5
=> [1, 2, 3, 4, 5]
[gameParser]: my_list.remove(4)
=> 5
[gameParser]: my_list
=> [1, 2, 3, 4]
[gameParser]: my_list.remove()
=> 4
[gameParser]: my_list
=> [1, 2, 3]
[gameParser]: my_list.insert(0, "a")
=> ["a", 1, 2, 3]
[gameParser]: my_list.len
=> 4
```

Figur 15: List-operatorer

In-och utmatning

För inmatning används **read()** där en sträng kan skickas med som argument. Detta skriver eventuell sträng till terminalen och inväntar input från spelaren som sedan returneras med ENTER (se figur 16).

```
[Gameparser]: Player_Name = read("What is your name?")
What is your name?
Generic Name
=> Generic Name
```

Figur 16: Inmatning

Utmatning hanteras av **write()** som enbart skriver en sträng i terminalen. **write()** returnerar nul (se figur 17).

```
[Gameparser]: write("Hello" + Player_Name + "!")
Hello Generic Name!
=>
```

Figur 17: Utmatning

För att kunna skriva ut ett meddelande och en integer behövs en typ-konvertering från integer till sträng. Detta görs med hjälp av `str()`. Figur 18 visar hur en typ-konvertering utförs i Gameparser.

```
[gameParser]: score = 150
=> 150
[gameParser]: write("Your score is : " + str(score))
Your score is : 150
=>
```

Figur 18: Typ-konvertering från heltal till sträng

Villkor

Villkor kan definieras med **if** eller **if else**. **if** kräver ett villkor följt av ett block som exekveras om villkoret uppfylls. **else** är en frivillig bisats som kan vara bra som alternativ då villkoret ej uppfylls. Även här krävs ett block som körs då **else** uppfylls (se figur 19).

```
[Gameparser]: x = 10
=> 10
[Gameparser]: if x == 10 {true} else {false}
=> true
[Gameparser]: x = 10
=> 10
[Gameparser]: if x != 10 {true} else {false}
=> false
```

Figur 19: Exempel på hur en if-sats kan skrivas

För fler alternativ kan **switch** användas där en variabel jämförs med flera **cases** för att avgöra vilket block som exekveras (se figur 20).

```
[Gameparser]: x = 1
=> 1
[Gameparser]: switch (x) case(1) {"one"} case(2) {"two"}
=> "one"
```

Figur 20: Exempel på hur en switch-sats kan skrivas

Repetitionssatser

Det finns två verktyg vilka kan användas för att implementera iteration i språket. Med en **while**-loop är det möjligt att skapa en repetitionssats som repeteras så länge ett villkor uppfylls. En **for**-loop itererar över en lista eller en range. Figur 21 visar ett exempel på en while-loop och figur 22 visar ett exempel på en for-loop.

```
[gameParser]: limit = 0
=> 0
[gameParser]: while limit < 4 { limit = limit + 1 }
=>
[gameParser]: limit
=> 4
```

Figur 21: Exempel på while-loop


```
[gameParser]: for number in (1..5) { write(number)}
1
2
3
4
5
[gameParser]: list = ["x", "y", "z"]
=> ["x", "y", "z"]
[gameParser]: for element in list { write(element) }
x
y
z
```

Figur 22: Exempel på for-loop

Break

Med nyckelordet **break** kan en loop avbrytas oavsett om villkorssatsen är uppfylld eller inte (se figur 23). Oftast är det lämpligt att ha en **if**-sats för att se om ett mål har uppnåtts och om det är så används **break** för att avsluta iterationen. **break** kan vara mycket användbar när det gäller oändliga loopar.

```
while true
{
  break
  write("Hello")
}
```

Figur 23: While-loopen avslutas utan någon utskrift

I figur 24 visas ett praktiskt exempel på hur en **break** kan vara användbar när det gäller en oändlig while-loop. Figur 25 visar att det går lika bra att använda **break** i en range-baserade for-loop.

```
counter = 0
while true
{
  if counter == 5
  {
    write("The goal has been achieved")
    break
  }

  counter = counter + 1
}
```

Figur 24: Break inuti if-satsen avslutar while-loopen

```
for i in (1..100)
{
  if i == 5
  {
    write("The goal has been achieved")
    break
  }
}
```

Figur 25: Break inuti if-satsen avslutar for-loopen

Funktioner

Gameparser har stöd för användardefinierade funktioner. I figur 26 ser ni ett exempel på en enkel funktion som beräknar summan av två parameter och returnerar resultatet.

```
def sum(a, b)
{
    a + b
}

sum(1, 2)
```

Figur 26: En funktion som returnerar summan av två parameter (returnerar 3 i det här fallet)

Funktioner som definieras i programmet är tillgängliga globalt och är inte beroende på scope. Språket däremot inte tillåter att en funktion definieras inuti en annan funktion (nästling). Figur 27 visar ett otillåtet fall som ska undvikas när det gäller definition av funktioner.

```
def sum(a, b)
{
    def calculate(a, b)
    {
        a + b
    }
    calculate(a, b)
}
```

Figur 27: En funktion får **inte** definieras i en annan funktion i Gameparser

Rekursion

Stöd för rekursion låter programmeraren anropa en funktion inuti definitionen om och om igen. Med hjälp av rekursion löses vissa problem mycket effektivare jämfört med andra metoder och därför har detta implementerats i Gameparser. Figur 28 visar implementationen av funktionen "Fib" som räknar motsvarande Fibonacci-tal.

```
def Fib(n)
{
    if (n == 1) or (n == 2)
    {
        1
    }
    else
    {
        Fib(n-1) + Fib(n-2)
    }
}

Fib(30)    # Returnerar 832040
```

Figur 28: Funktionen som beräknar Fibonacci-talet för ett angivet tal

Enkla klasser

Gameparser erbjuder två konstruktioner som skiljer det från andra liknande språk. Dessa verktyg är avsedda för att göra det enklare för programmeraren att skapa ett textbaserat spel utan att t.ex. behöva skriva så många nästlade if-satser.

event

Den här konstruktorn motsvarar ett vägsål där spelaren ges alternativ med vilket sedan programmet körs vidare. Ett event kan exempelvis representeras som ett skog där spelaren får veta om situationen där och får välja vad denne ska göra. Ett event består av två huvuddelar: **init** och **run**. Init är till för att initiera event. Där kan lokala variabler initieras med ett önskad startvärde. All kod inuti init körs **en gång** när spelet för första gången anropar eventet. Run-delen är tänkt att vara kroppen av eventet där programmeraren kan bestämma vad som ska hända. Till exempel är det möjligt att skriva ut lite text för att informera spelaren om situationen. Sedan kan inmatningen läsas med hjälp av read och en villkorssats kan bestämma hur programmet ska ta sin väg till en lämplig konsekvens.

Det är ganska fritt i init/run-delen när det gäller kod. Programmeraren har i stort sett tillgång till nästan alla konstruktioner såsom repetitionssatser, villkorssatser, funktionsanrop, tillgång till globala variabler o.s.v. Distinktionen är att init initierar eventet. Däremot är det inte tillåtet att definiera en funktion eller en annan klass inuti ett event. Figur 29 visar ett enkelt exempel på hur en eventklass kan konstrueras.

```

event forest
{
  init
  {
    opened_chest = false
    write("You are entering a dark forest for the first time")
  }

  run
  {

    write("Dark forest")

    write("1. Walk forward")
    if opened_chest == false
    {
      write("2. Open the chest.")
    }

    choice = read()

    switch(choice)
    case(1)
    {
      # corresponding consequence
    }
    case(2)
    {
      # corresponding consequence
    }
  }
}

```

Figur 29: Implementering av ett event

Efter att ett event har skapats går det att ladda eventet med hjälp av den reserverade funktionen **load()**. Observera att ett event inte beroende på scope och kan kallas var som helst i koden. **load()** tar ett event som argument och eventuellt filnamnet på en bild (jpg eller png) placerad i images-mappen (se figur 30). Denna bild visas då **load()** körs. Bilden stängs med Escape.

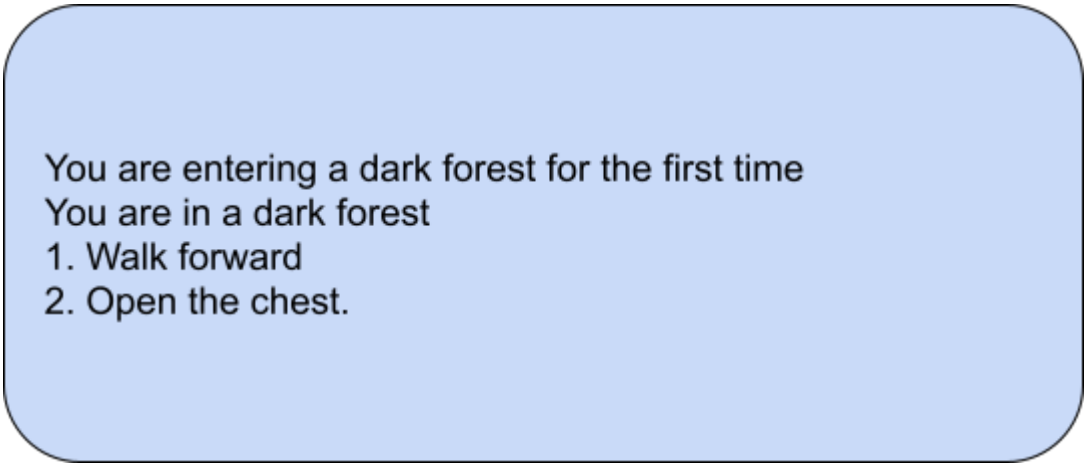
```

load(forest)
load(forest, "forest.jpg")

```

Figur 30: load() används för att köra ett event

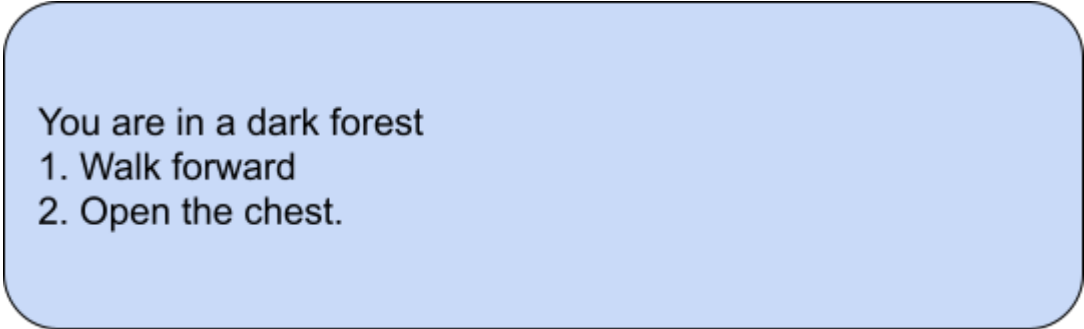
Resultatet av första körningen visas i figur 31.



You are entering a dark forest for the first time
You are in a dark forest
1. Walk forward
2. Open the chest.

Figur 31: Visar hur event "forest" ser ut när den körs för första gången

Notera att om eventet laddas för andra gången skrivs inte första raden ut som finns i init-delen (se figur 32).



You are in a dark forest
1. Walk forward
2. Open the chest.

Figur 32: Visar hur event "forest" ser ut när den körs för andra gången

prop

Denna konstruktor används för att skapa abstrakta objekt som kan innehålla en eller flera datamedlemmar. Det finns ingen run-del, istället det går att definiera en parameterlista för att sedan kunna skapa en instans av klassen med godtyckliga värden.

En instansiering av en prop görs genom att använda nyckelordet "**new**". Efter att en instans har skapats går det att komma åt dess datamedlemmar med en punkt. Figur 33 visar ett exempel på definition samt instansiering av ett **prop**.

```

prop character
{
  init(n, hp, att, agi)
  {
    name = name
    health = hp
    attack = att
    agi = agility
  }
}

player = character.new("Kratos", 100, 25, 8)
write( player.name )      # "Kratos" skrivs ut

enemy = character.new("Skeleton, 10, 3, 2)
write( enemy.name )       # "Skeleton" skrivs ut

```

Figur 33: Exempel på hur ett prop kan definieras och instansieras

Notera att oftast behöver ett spelar-objekt nås i olika event och funktioner i spelet. Därför är det lämpligt att spelar-objektet deklarereras globalt (se figur 34).

```

$player = character.new("Kratos", 100, 25, 8)
write( $player.name )      # "Kratos" skrivs ut

```

Figur 34: Spelar-objektet skapas med fördel som globalt

Scope

Scope hanteras internt via hashtabeller i en lista. Varje scope har ett heltal som korrelerar med en av hashtabellerna där dess variabler lagras. En variabel i t.ex. en funktion kan därför inte nås från ett annat scope (se exemplet i figur 35). För detta krävs globala variabler.

```
[Gameparser]: def func() { x = 7 write(x) }
=>
[Gameparser]: func()
7
=>
[Gameparser]: x
=> (crash) Unknown variable 'x'
```

Figur 35: Funktioner har egna lokala variabler som inte är tillgängliga utanför funktions-blocket

Samtliga funktionsvariabler är definierade inom funktions-ramen mellan måsvingar och programmeraren har friheten att använda och ändra variabeln utan att den påverkar andra variabler med samma namn i ett annat scope. Figur 36 visar ytterligare exempel på detta.


```

def fun1(x)
{
    write(x)
}

def fun2(x)
{
    x = x + 10
    fun1(x)
}

def fun3(x)
{
    x = x + 10
    fun2(x)
}

x = 10
fun1(x)           # resultatet är 10
fun2(x)           # resultatet är 20
fun3(x)           # resultatet är 30
write(x)          # resultatet är 10

```

Figur 36: Scope-hantering i Gameparser

Det gäller samma princip för definitionen av ett event och prop och deras variabler och data medlemmar kan inte kommas åt från ett annat scope.

Andra fördefinierade funktioner

Förutom de fördefinierade funktioner som förklarades ovan (**write()**, **read()**, **load()** och **str()**) finns det två ytterligare funktioner som kan vara användbara när det gäller textbaserad spelprogrammering:

- **wait()**: som är för att pausa körningen av spelet för ett antal sekunder
- **cls()**: som rensar terminalen.

4. Systemdokumentation

Lexer

Den lexikaliska analysen bygger på att *rdparse* delar upp inläst kod i “tokens” utifrån reguljära uttryck. Först rensas koden på kommentarer och blanktecken. Sedan identifieras reserverade ord samt alla tillgängliga datatyper. Dessa tokens lämnas till sist över till parsern.

Parser

Gameparser använder *rdparse* för sekventiellt bearbeta inlästa tokens. De sammankopplas till ett abstrakt syntaxträd.

Abstrakt syntaxträd

Nu består programmet av långa grenar rotade ur programmet som helhet. Varje spets har specificerats till en datatyp. Varje förgrening kan ses som en nod där ett objekt skapas med en init- och en eval-funktion. De två verktygen låter parsern först bygga upp trädet och sedan exekvera koden strukturerat. Retursatser klättrar längs med grenarna tills slutligen den sista kodraden bearbetats.

Kodstandard

Det finns ingen specificerad kodstandard utöver de krav som tidigare nämnts. Block skapas inom måsvingar, variabelnamn inleds med bokstäver och globala variabler inleds med \$-tecken.

BNF

<prog>	::=	<comps>
<comps>	::=	<comps> <comp> <comp>
<comp>	::=	<definition> <statement>
<definition>	::=	<prop> <event> <function_def>
<prop>	::=	prop <i>Identifier</i> { <init> }
<event>	::=	event <i>Identifier</i> { <init> run <block> }
<init>	::=	init (<params>) <block> init <block>
<function_def>	::=	def <i>Identifier</i> (<params>) <block>
<params>	::=	<params> , <param> <param> <empty>
<param>	::=	<identifier>
<block>	::=	{ <statements> } { <empty> }
<function_call>	::=	<i>Identifier</i> (<values>) write (<i>Identifier</i>) write (<i>GIdentifier</i>) write (<exp>) write () read (<exp>) read () wait (<i>Integer</i>) load (<i>Identifier</i> , <exp>) load (<i>Identifier</i>) str (<exp>) cls ()

<statements>	::=	<statements> <statement> <statement>
<statement>	::=	<condition> <loop> <assignment> <array_op> <exp> <i>Break</i>
<assignments>::=		<assignments> <assignment> <assignment>
<assignment>	::=	<identifier> = <array_op> <identifier> = <exp> <identifier> [<i>Integer</i>] = <exp> <identifier> . <i>Identifier</i> = <exp>
<exp>	::=	<bool_exp> and <exp> <bool_exp> or <exp> not <exp> <bool_exp>
<bool_exp>	::=	<math_exp> <i>CompOp</i> <math_exp> <bool_val> <i>CompOp</i> <bool_val> <bool_val>
<bool_val>	::=	true false <math_exp>
<math_exp>	::=	<math_exp> + <term> <math_exp> - <term> <term>
<term>	::=	<term> * <factor> <term> / <factor> <factor>
<factor>	::=	<i>Integer</i> <signs> <i>Integer</i> <signs> (<math_exp>) (<exp>) <function_call> <array>

		<instancing> <instance_reader> <variable_node> <i>LiteralString</i> <i>Range</i>
<signs>	::=	<signs> <sign> <sign>
<sign>	::=	+ -
<array>	::=	<identifier> [<i>Integer</i>] [<values>] []
<array_op>	::=	<identifier> <operations> <array> <operations> <instance_reader> <operations> <:function_call> <operations>
<operations>	::=	<operations> <operation> <operation>
<operation>	::=	[<i>Integer</i>] . remove (<exp>) . remove () . append (<exp>) . append << <exp> . insert (<i>Integer</i> , <exp>) . len
<values>	::=	<values> , <exp> <exp> <empty>
<instancing>	::=	<identifier> . new (<values>)
<instance_reader>	::=	<identifier> . <identifier>
<condition>	::=	<if> <switch>
<if>	::=	if <exp> <block> else <block> if <exp> <block>

```

<switch>      ::=  switch ( <exp> ) <cases>

<cases>       ::=  <cases> <case>
                   | <case>

<case>        ::=  case ( <exp> ) <block>

<loop>        ::=  while <exp> <block>
                   | for Identifier in Range <block>
                   | for Identifier in <array> <block>
                   | for Identifier in Identifier <block>
                   | for Identifier in GIdentifier <block>

<identifier>  ::=  Identifier
                   | GIdentifier

<variable_node> ::= Identifier
                   | GIdentifier

```

5. Reflektion

Projektet lyckades utan större svårigheter hålla sig inom de uppsatta tidsramarna. Den del som främst utmärkte sig var då scope och funktioner skulle implementeras. Dessa element visade sig vara sammanbundna och därmed till stor del ömsesidigt beroende.

Originalplanen var inte att implementera funktioner men allt eftersom projektet fortgick blev behovet av dessa uppenbart. Detta krävde en del arbete men när det väl var genomfört gick implementationen av rekursion smidigt. Det som var svårt med funktionsimplementationen var att det kändes förvirrande att skapa klasser och noder för senare användning. Körningen av kod i run-time ledde till att programmet inte kördes pålitligt och eftersom testningen i början främst fokuserat på interaktivt läge märktes det inte att kod med flera rader inte lyckades köras. Lösningen var att bryta ut kod som ska köras till evaluate-delen i motsvarande klass.

När dessa hinder övervunnits fann projektet sig plötsligt framför utlagt tidsplan vilket lämnade rum för mer finputsning i slutskedet.