TabTab

Ett programmeringsspråk i Ruby
v 1.2

Joakim Bodin & Kevin Nortier

Linköpings Universitet
2010-05-25

Innehållsförteckning

1. Inledning

Följande projekt är en del av kursen TDP019 på programmet Innovativ Programmering. Målet med projektet är att skapa ett eget programmeringsspråk, anpassat till en viss målgrupp. Syftet med detta är att få djupare förståelse för hur ett språk är uppbyggt.

Vi valde att implementera ett språk som är en blandning av Python och C++. Språket riktar sig till programmerare som har programmerat förut, men inte är erfarna ännu. Det kan ses som ett övergångsspråk mellan Python och C++, då tvingad indentering ger vanan att skriva "städad" kod – men det introducerar också koncept som statiskt typning. Språket kan även användas av programmerare som föredrar statisk typning från C++ men gärna skriver med tvingad indentering från Python.

2. Användarhandledning

Följande är en användarhandledning för TabTab.

2.1 Introduktion

TabTab är ett språk för enkel imperativ programmering. Hela programmet ska vara omslutet av nyckelorden "program" och "endprogram", och all kod däremellan måste vara konsekvent indenterat. Alla TabTab-filer har en filändelse på ".tt". Ett kort program i TabTab skulle kunna se ut så här:

```
program

     fun void max(int x, int y)
          if(x > y)
               output x
          else
               output y
          endif
     endfun

     int distance = 1337
     int time = 42

     max(distance, time)

endprogram
```

Programmet ovan skapar en funktion "max", som skriver ut det högsta av dom två värden som den får in som parametrar. Vi deklarerar två variabler med namnen "distance" och "time", och skickar sedan in dom till funktionen.

2.2 Variabler och typer

Variabeltilldelning i TabTab är statiskt typad, vilket innebär att man alltid måste ange vilken typ värdet på en variabel har när man deklarerar denna. Värdet som tilldelas måste också stämma överens med typen som angetts.

Dom typer som finns i TabTab är <int>, <float>, <bool>, <string> och <array>. Det finns även en typ <void>, men den används endast för returvärden i en funktionsdefinition (se avsnittet om funktioner). För att deklarera en variabel som heter "number" med värdet "42" skriver man:

```
int number = 42
```

Typen <array> är lite speciell. En <array> måste deklareras på så sätt att man skriver vad den ska innehålla. En <array> som heter "values" och ska innehålla ett godtyckligt antal <string> element deklareras på följande sätt:

```
string array values = ["first", "second", "third"]
```

En <int> är vilken sifferkombination som helst, utan decimaltecken. Siffror med decimaltecken definieras istället som <float>.

En <bool> kan vara antingen "true" eller "false" (utan citattecken).

En <string>, eller sträng, är en kombination av tecken omslutna av enkla eller dubbla citattecken, till exempel är både "value1" och 'value1' godkända strängar.

En <array> är som demonstrerat ovan en lista med element, separerade av kommatecken och omslutna av hakparanteser, [].

Typen <void> har inget värde alls utan används bara för att definiera en funktion utan returvärde.

Efter att en variabel har deklarerats, kan man ändra dess värde hur man vill så länge typen är korrekt. Ett exempel på sådan användning:

```
float current = 87320.0
float earned = 48151.0
float spent = 2342.0

current = current + (earned - spent)
```

Observera att variabelnamn inte får heta något av dom nyckelord som används av konstruktioner i TabTab. Variabelnamn kan dock innehålla eller delvis bestå av upptagna nyckelord. Nyckelorden som används i TabTab är:

program	while	and	void
endprogram	endwhile	or	int
fun	for	not	float
endfun	in	true	array
if	endfor	false	string
else	input	break	bool
endif	output	skip	return

2.3 Funktioner

En viktig del av alla programmeringsspråk är funktioner. En funktion består utav kod som endast körs när man kallar på funktionen. Den kan ta in parametrar och returnera värden.

I TabTab kan en funktionsdefinition se ut på följande sätt:

```
fun void min(int x, int y)
     if(x < y)
          output x
     else
          output y
     endif
endfun
```

En funktionsdefinition börjar och slutar med nyckelorden "fun" och "endfun". Innehållet är indenterat. Om funktionen ska returnera ett värde, måste man ange vilken typ det returvärdet kommer att ha. I exemplet returnerar vi ingenting, returvärdet representeras av <void>. Sedan kommer namnet på funktionen, som vi angett till "min" och vilka typer av parametrar som måste skickas med vid anrop. I det här fallet kräver "min" att man skickar med två objekt av typen <int> när man anropar funktionen.

Vid funktionsanrop gäller det att alla parametrar skickas med:

```
int length = 7
int height = 14

min(length, height)
```

I exemplet ovan kommer funktionen "min" att köras med våra två värden, och kommer i det här fallet att skriva ut värdet "7".

Skulle en funktion sakna inparametrar i sin definition, skriver man tomma paranteser vid funktionsanrop. Funktionsanrop måste ske efter att funktionen redan definierats, det går inte att kalla på en funktion tidigare i koden än där den har skrivits.

En funktion kan också returnera ett värde. I TabTab gör man detta genom att skriva till exempel:

```
return value
```

Ovan exempel skrivs i en funktion, där "value" är namnet på en tidigare deklarerad variabel. Observera att funktionen måste ha definierats som att den ska returnera ett värde av den typen om det ska gå att returnera ett värde. När man använder "return" avslutas funktionen, och resterande kod i funktionen körs inte. Returvärdet kan sedan sparas i en variabel genom att skriva till exempel:

```
int product = times(3, 7)
```

Fallet ovan är en variabeldeklaration med namnet "product", och "times" är en funktion som får inparametrar "3" och "7". Returvärdet från funktionen "times" sparas i variabeln "product".

2.4 Input och output

För att få in information från användaren eller få ut information till användaren har vi "input" respektive "output". Nyckelordet "input" kommer att invänta information från användaren, och den informationen kan sparas undan i en variabel. Om vi till exempel vill ha användarens namn:

```
output "Enter your name:"
string name = input
output "Your name has been saved as:", name
```

Konstruktionen "output" kan användas för att skriva ut flera uttryck separerade av kommatecken som illustreras ovan.

Input kan endast användas för datatypen <string>.

2.5 If-satser

En grundläggande konstruktion i många språk är "if"-satsen. I TabTab är uppbyggnaden av en if-sats väldigt enkel. En if-sats skulle kunna se ut så här:

```
if(x == y)
     output "Match!"
else
     output "No match!"
endif
```

Idén är att om villkoret "x == y" är sant, exekveras grenen direkt under if-satsen. I det här fallet skulle vi skriva ut "Match!". Om villkoret däremot är falskt, hamnar vi i "else"-grenen, och vi skulle istället skriva ut "No match!". If-satser kan nästlas i varandra, och varje påbörjad if-sats måste avslutas med nyckelordet "endif". Else-grenen behöver inte ens existera, om man vill att ingenting ska hända när uttrycket är falskt. Observera att innehållet i en if-sats måste indenteras korrekt.

En if-sats kan bli betydligt mer komplicerad än exemplet ovan. Villkoret som måste bli sant kan delas upp med ett antal olika operatorer:

Logisk "and": (x and y) – både x och y måste vara sanna för att uttrycket ska bli sant.

Logisk "or": (x or y) – om x eller y är sann, blir uttrycket sant (även om båda är sanna).

Logisk "not": (not x) – om x är falskt, blir uttrycket sant.

Jämförelse "<": (x < y) – om x är mindre än y, blir uttrycket sant.

Jämförelse ">": (x > y) – om x är större än y, blir uttrycket sant.

Jämförelse ">=": (x >= y) – om x är större än eller lika med y, blir uttrycket sant.

Jämförelse "<=": (x <= y) – om x är mindre än eller lika med y, blir uttrycket sant.

Jämförelse "==": (x == y) – om x är lika med y, blir uttrycket sant.

Jämförelse "!=": (x != y) – om x inte är lika med y, blir uttrycket sant.

Dessa jämförelser kan kombineras för att skapa komplexa uttryck. Paranteser kan användas för att ändra prioritet.

2.6 For-loopar

För att iterera genom en "container", det vill säga en "array" eller "string", kan man använda sig utav en "for"-loop. Ett exempel på en for-loop skulle kunna se ut så här:

```
string name = "Bob"
for (x in name)
     output x
endfor
```

Ovan kod kommer att skriva ut alla bokstäver i strängen en och en. Variabeln "x" kommer alltså att vara nuvarande element i varje varv av loopen. I första varvet är "x" lika med "B", i andra varvet "o", och i sista "b". Variabeln behöver inte heta "x", utan den kan ha vilket godkänt "identifier"-namn som helst.

För att förtydliga detta kan vi demonstrera ett exempel som loopar igenom en array. Säg att vi vill se om en array innehåller värdet "7":

```
int array values = [42, 1337, 7]

for (num in values)
     if (num == 7)
          output "Match found!"
          break
     else
          skip
     endif
endfor
```

Loopen kommer att sätta variabeln "num" först till "42" och göra jämförelsen i if-satsen. När den ser att det är falskt, kommer den in i "else"-grenen. I else-grenen använder vi nyckelordet "skip", som går vidare till nästa varv i loopen. När "num" får värdet "7" skriver vi istället ut "Match found!" och avbryter loopen med "break".

Eftersom det inte händer mycket i just den här for-loopen kan "skip" tänkas onödig, men den är mycket användbar i andra situationer. Nyckelordet "break" kan också användas, men den hoppar ut ur loopen helt och kör inga fler varv.

Se kapitel 2.7 While-loopar för ytterligare information om loopar.

2.7 While-loopar

Ett annat sätt att skapa loopar på är att använda en "while"-konstruktion. En while-loop fortsätter att köra sin kod om och om igen tills villkoret blir falskt. En enkel while-loop kan skrivas på följande sätt:

```
x = 100
while(x > 0)
     output x
     x = x - 1
endwhile
```

Konstruktionen omsluts av nyckelorden "while" och "endwhile", med innehållet korrekt indenterat. Denna loop kommer att skriva ut alla tal från 100 till 1 och sedan avslutas. En while-loop skulle kunna beskrivas som en upprepande if-sats där else-grenen avslutar loopen.

En viktigt punkt angående while-loopar är att en variabeldeklaration i loopen kommer att temporärt skriva över tidigare deklarationer av en variabel med samma namn.

```
int y = 100
int x = 100
while(x > 0)
     int y = 42
     output y
     x = x - 1
endwhile
```

I exemplet ovan kommer "42" att skrivas ut 100 gånger. Variabler som deklareras i while-loopens egna kod kommer att prioriteras, och endast om man refererar till en variabel som inte deklarerats i loopen kommer while-loopen att leta efter en variabel utanför den egna loopen.

Variabeln "y" som deklarerades utanför loopen kommer alltså att vara oförändrad, eftersom while-loopen skapade sin egen variabel "y". Variabeln "x" finns inte deklarerad i while-loopen, alltså används och ändras värdet på "x" utanför loopen.

När loopen är avslutad försvinner även variabler som deklarerats.

2.8 Körning

TabTab körs med fördel i Linux-miljö. För att köra koden som skrivits i en ".tt"-fil, måste man först kontrollera att Ruby är installerat. Detta kan kontrolleras i kommandotolken genom att skriva "ruby -v", och då ser man vilken version av Ruby som är installerat.

Kontrollera sedan att alla relevanta TabTab-filer finns. Filerna som krävs är:

1. tabtab
2. RDParser.rb
3. tabtest.rb
4. TTParser.rb

Körningen av koden är sedan väldigt enkel. Säg att vår TabTab-fil heter "myprogram.tt". För enkelhetens skull ligger filen i samma mapp som TabTab-filerna. Körning av programmet skulle då se ut så här:

```
>> ruby tabtab
Enter file path:
myprogram.tt
```

Om filerna skulle ligga i olika mappar måste specifik plats anges:

```
>> ruby ~/user/tt/tabtab
Enter file path:
~/user/files/myprogram.tt
```

3. Erfarenheter och reflektion

När vi skrev grammatiken tyckte vi att allt passade ihop väldigt bra, vi såg inga uppenbara hål i vår logik. När det blev dags att implementera parsningen med hjälp av en RDParser upptäckte vi en del praktiska problem som delvis berodde på att vi använde just RDParsern. Vi kunde inte göra upprepningar av uttryck på det sättet vi hade tänkt (genom att använda reguljära uttryck), vi fick istället skriva oss runt det genom att skapa ytterligare regler som en matchning var tvungen att gå igenom. Med hjälp utav detta kunde vi få en sorts rekursion som gjorde att vi kunde matcha oändligt långa listor om vi ville, vilket är ett måste eftersom vi inte vet i förväg till exempel hur många satser det ingår i ett program.

Ett liknande problem uppstod med vår matematik. RDParsern "prioriterade" från högerled, och med en kombination av multiplikation och addition blev resultatet inte som vi hade väntat oss. Det löste vi genom att skapa ny regler som innan, och matchningen var tvungen att gå igenom flera regler för att nå rätt. Med det lyckades vi se till att multiplikation och division fick högre prioritet än addition och subtraktion, och vi kunde även tvinga RDParsern att "prioritera" vänsterled istället för högerled. Idén till detta kom delvis från vår lösning med upprepade uttryck, och fullbordades när vi studerade Pythons grammatik för att se hur den löste problemet.

Det tog ett tag rent allmänt att förstå hur RDParsern fungerade och vi var tvungna att ta en del omvägar för att den inte betedde sig som vi hade förväntat oss. Ett exempel på detta var när våra nyckelord som "if", "while" och "for" matchades som <identifiers> istället vad dom skulle matchas till. Vi tyckte att RDParsen borde matcha rätt efter våra matchningsregler, men det gjorde den inte. Vi fick till slut skriva ett väldigt långt reguljärt uttryck som beskrev vad en <identifier> fick vara och inte fick vara. Vi blev tvungna att inkludera alla nyckelord som skulle kunna vara en godkänd <identifier> men inte fick vara det. Till exempel "program", "return", "output" och alla andra konstruktioner vi har. Det löste problemet men vi förstår fortfarande inte riktigt hur RDParsern prioriterar ibland.

RDParsern vi använde hade lite annorlunda utskrifter också. Den var ju skriven för ett exempel med tärningskast som vi hade i en tidigare kurs, och därför var den inte riktigt anpassad till att hantera tokens som var större än en karaktär. Det var inget som förstörde programmet men spårutskrifterna var väldigt svåra att förstå då dom hämtade ut delar av tokens som inte alls hade med det utskriften handlade om att göra. Vi ändrade därför i RDParsern att vi kunde använda spårutskrifterna för felsökning.

En sak som vi inte riktigt uppskattar i Ruby är att Ruby alltid vill returnera något. Det leder till att våra utskrifter alltid har "nil" i slutet och ibland har vi även fått lite konstigare fel som vi inte förstod till en början. Det kunde till exempel vara att programmet skrev ut en hel array. Det visade sig att det var för att vi hade skrivit något längst ner i en funktion som vi inte alls ville returnera men att Ruby gjorde det ändå. Vi skulle gärna se att det fanns något sätt att göra sig av med det där extra "nil" som returneras oavsett vad vi gör.

När vi implementerade "break" och "return" ville vi ha något samlingsnamn på dom eftersom dom beter sig på liknande sätt i vårt språk. Vi valde då att ge en variabel namnet "abort", det kändes som om det var ungefär det som "break" och "return" gjorde. Det vi inte visste var att det är ett reserverat ord i Ruby som verkar avbryta tolken på något sätt, och kraschade vårt program. Det tog ganska

mycket tid innan vi insåg att det var variabelnamnet som var problemet, eftersom vi hade ändrat på en hel del efteråt. Vi fick ju heller inget specifikt felmeddelande om att det var just variabelnamnet som var problemet, vilket gjorde det betydligt svårare att felsöka.

Vi hade en del problem med att ha mer än ett "statement", och experimenterade en del med det innan vi fick det att fungera. Problemet var att vi hade radbrytningar med i grammatiken, och det översattes inte väl i praktiken och RDParsern. Vi var tvungna att tänka om hur vi faktiskt skulle hantera radbrytningar. Fler problem uppstod när vi hade tomma rader emellan (och därmed fler radbrytningar). Vi löste det till slut genom att ändra i reglerna hur och var radbrytningarna skulle finnas, och sedan gjorde vi också så att vi satte ihop alla tomma rader mellan "statements" till bara en radbrytning.

Problemet med radbrytningar flöt in på problemet vi hade med att ha kommentarer på samma rad som en bit kod. Det löstes delvis av problemet ovan, men vi gjorde också så att vi stoppade in en matchning i reglerna där ett "statement" kunde följas av en "comment", och sen gjorde vi ingenting med den kommentaren vi hittade.

Vi hade även planer på att konvertera strängar vi fick in från "input" till motsvarande "int", "float", "bool" eller "array" beroende på vad variabeln hade för typ. Det visade sig vara lite väl optimistiskt eftersom vi var tvungna att skriva reguljära uttryck och gå igenom ett antal operationer för att verkligen få det att fungera ordentligt. Tiden räckte inte till och "input" fick bli begränsad till att bara ta in strängar.

4. Bilagor

4.1 Systemdokumentation

4.1.1 Översikt

TabTab består utav en RDParser, en intenderings-check och implementationen av själva språket.

När man skickar in en fil till TTParsern kollas först om indenteringen är korrekt. Är den inte det, returneras ett felmeddelande som anger på vilken rad det är fel, hur många indenteringstecken som förväntas och hur många som hittades. Ett indenteringstecken kan vara två eller flera mellanslag, eller ett eller flera tab-tecken. Det som kommer först blir standarden för resten av dokumentet.

Går koden igenom indenteringskontrollen används RDParsern för att hämta ut lämpliga tokens. Dessas tokens matchar sedan regler (en komplett lista på regler finns i sektion 3.1 Grammatik och tokens i 3.3 Tokens). Reglerna matchas på sådant sätt att programmet kan bygga upp ett syntaxträd, och skapa objekt på lämpliga ställen i koden. Ett objekt kan i det här fallet vara till exempel en lista med "statements", som i sin tur innehåller ett eller flera "statement", där ett statement kan vara en if-sats eller liknande konstruktion.

Börjar vi på toppen av trädet ser vi från grammatiken att vi behöver nyckelorden "program" och "endprogram" med en lista av statements däremellan. Statements innehåller i sin tur ett eller flera statement, och vi går sedan djupare och djupare till rötterna av det här trädet, tills vi hamnar på dom minsta byggstenarna som siffror, nyckelord och operatorer. Trädet byggs upp genom att identifiera dessa minsta byggstenar och sedan sätta ihop dom på ett sådant sätt att vi till slut hamnar i toppen av trädet igen.

Från den konstruktionen kan vi sedan kalla på en funktion som tillhör vår statements-lista. Funktionen heter "eval", och motsvarande funktion finns för alla delar av trädet. Vi kallar på den funktionen i toppen utav trädet för att köra koden, och den körs då får alla beståndsdelar ända ner till våra minsta byggstenar.

4.1.2 Kodstandard

Den kodstandard som använts är följande:

- Variabelnamn skrivs med små bokstäver, och separeras med "_" där orden separeras i vanliga fall.
- Funktionsnamn har samma standard som en variabel.
- Klassnamn börjar med stor bokstav, resten av tecknen är små och separeras med "_" likt variabler och funktioner.
- Konstruktioner separeras med radbrytningar för läsbarhet, helst bara en tomrad emellan.

4.1.3 Klasser

TabTab har flera klasser i språket som man kan kalla för en "gate". En gate håller ihop alla beståndsdelar under sin egen konstruktion. Till exempel finns det en gate för en if-sats, och den innehåller då villkoret som krävs för att if-satsen ska gå igenom (expression), listan med statements som ska exekveras och eventuell else-gren med motsvarande statements-lista. En instans av denna gate skapas när en if-sats har hittats i koden, och är det också den som gör själva kontrollen i Ruby för att välja vilken gren som faktiskt ska köras.

Motsvarande klasser finns för samtliga konstruktioner i TabTab (se sektion 2. Användarhandledning).

4.1.4 Paketering

TabTab kan användas i en miljö som har Ruby installerat. Filerna som ingår i TabTab är:

1. tabtab
2. RDParser.rb
3. tabtest.rb
4. TTParser.rb

I Linux-miljö används med fördel kommandotolken för att ladda in en ".tt"-fil i "tabtab", som sedan använder sig utav "TTParser.rb", "RDParser.rb" och "tabtest.rb" (för en mer detaljerad förklarling, läs 2.8 Körning).

4.1.5 Tokens

När parsningen av koden sker i RDParsern finns det ett antal tokens vi hämtar ut för att kunna matcha med reglerna för TabTab. Vi matchar tokens genom att använda Rubys reguljära uttryck. Dom tokens vi valt att hämta ut är följande:

```
#Newline
token(/\s*\n+/) {"\n"}
```
Hämtar ut alla radbrytningar. Finns det mellanrum innan radbrytningen, returneras bara radbrytningen.

```
#Comment
token(/\/\/.*\s*\n+/) {|m| m}
```
Hämtar ut alla kommentarer och returnerar dessa. Kommentarer räknas inte som körbar kod, men matchas ändå som en del av vissa regler för att man till exempel ska kunna skriva kommentarer direkt efter en rad kod. Tar även bort extra mellanrum och tomrader efter en kommentar.

```
#Remove whitespace
token(/\s+/)
token(/ /)
```
Matchar alla mellanslag, radbrytningar och tab-tecken för att sedan inte göra någonting med dom. Kommer man till token-matchning är indenteringen redan korrekt och behöver därför inte användas mer.

```
#Float
token(/\d+\.\d+/) {|m| m.to_f}
#Int
token(/\d+/) {|m| m.to_i}
```
Matchar alla tal, med eller utan decimaltecken.

```
#String
token(/"[^"]*[^\\]"/) {|m| m}
token(/'[^']*[^\\]'/) {|m| m}
```
Matchar alla strängar, det vill säga allt som är inom enkla eller dubbla citationstecken.

```
#Parenthesis
token(/[\(\)]/) {|m| m}
```
Matchar paranteser, men ej innehållet däremellan.

```
#Calc Operators
token(/\+/) {|m| m}
token(/-/) {|m| m}
token(/\*/) {|m| m}
token(/\//) {|m| m}
```
Matchar alla godkända matematiska operatorer i språket.

```
#Comp Operators
token(/<=/) {|m| m}
token(/>=/) {|m| m}
token(/==/) {|m| m}
token(/</) {|m| m}
token(/>/) {|m| m}
```
Matchar alla godkända logiska operatorer i språket.

```
#Comma
token(/,/) {|m| m}
```
Matchar ett kommatecken, för användning i språkets regler.

```
#Assignment Operator
token(/=/) {|m| m}
```
Matchar likhetstecknet.

```
#Array
token(/\[/) {|m| m}
token(/\]/) {|m| m}
```
Matchar hakparenteser, men inte innehållet däremellan.

```
#Word
token(/[a-zA-Z_][a-zA-Z0-9_]*/) {|m| m}
```
Matchar ord, nyckelord eller identifierare.

```
#Everything else
token(/.+/) {|m| m}
```
Matchar allt annat som är kvar, om det finns något över.

4.1.6 Grammatik

```
<program> ::=
      "program" "\n" <statements> "endprogram" "\n"

<def_fun> ::=
      "fun" <fun_type> <identifier> "(" <parameters> ")" "\n"
      <statements> "endfun"
      | "fun" <fun_type> <identifier> "(" ")" "\n" <statements>
      "endfun"
      | "fun" <array_type> <identifier> "(" <parameters> ")" "\n"
      <statements> "endfun"
      | "fun" <array_type> <identifier> "(" ")" "\n" <statements>
      "endfun"

<call_fun> ::=
      <identifier> "(" <arguments> ")"
      | <identifier> "(" ")"

<for_stmt> ::=
      "for" "(" <identifier> "in" <container> ")" "\n" <statements>
      "endfor"

<while_stmt> ::=
      "while" "(" <expression> ")" "\n" <statements> "endwhile"

<if_stmt> ::=
      "if" "(" <expression> ")" "\n" <statements> "endif"
      | "if" "(" <expression> ")" "\n" <statements> "else" "\n"
      <statements> "endif"

<comment> ::=
      /\/\/.*\n/

<def_var> ::=
      <type> <identifier> "=" <call_fun>
      | <identifier> "=" <call_fun>
      | <type> <identifier> "=" <expression>
      | <identifier> "=" <expression>
      | <type> <identifier> "=" <string>
      | <identifier> "=" <string>
      | <array_type> <identifier> "=" <array>
      | <identifier> "=" <array>

<container> ::=
      <string>
      | <array>
      | <variable>

<int> ::=
      Integer

<float> ::=
      Float


<array_elem> ::=
      <string>
      | <expression>

<array_elems> ::=
      <array_elems> "," <array_elem>
      | <array_elem>
```

```
<array> ::=
      "[" <array_elems> "]"
      | "[" "]"

<string> ::=
      /".*"/
      | /'.*'/

<bool> ::=
      <true>
      | <false>

<true> ::=
      "true"

<false> ::=
      "false"

<parameter> ::=
      <type> <identifier>
      | <array_type> <identifier>

<parameters> ::=
      <parameters> "," <parameter>
      | <parameter>

<argument> ::=
      <variable>
      | <expression>
      | <string>
      | <array>

<arguments> ::=
      <arguments> "," <argument>
      | <argument>

<expression> ::=
      <or_expr>

<para> ::=
      "(" <add_calc_expr> ")"

<num> ::=
      <para>
      | <int>
      | <float>
      | <variable>
<neg_expr> ::=
      <num>
      | "-" <num>

<multi_calc_expr> ::=
      <neg_expr>
      | <multi_calc_expr> "*" <multi_calc_expr>
      | <multi_calc_expr> "/" <multi_calc_expr>

<add_calc_expr> ::=
      <multi_calc_expr>
      | <add_calc_expr> "+" <multi_calc_expr>
      | <add_calc_expr> "-" <multi_calc_expr>

<calc_expr> ::=
      <add_calc_expr>
```

```
<comp_expr> ::=
      <calc_expr> <comp_operator> <calc_expr>
      | <calc_expr>
      | <bool>
      | <variable>

<comp_operator> ::=
      "<"
      | ">"
      | "=="
      | "!="
      | "<="
      | ">="

<not_expr> ::=
      <comp_expr>
      | "not" <not_expr>

<and_expr> ::=
      <not_expr>
      | <and_expr> "and" <not_expr>

<or_expr> ::=
      <and_expr>
      | <or_expr> "or" <and_expr>

<return> ::=
      <return> <expression>
      | <return> <string>
      | <return> <call_fun>
      | <return> <array>

<break> ::=
      "break"

<skip> ::=
      "skip"

<input> ::=
      "string" <identifier> "=" "input"
      | <identifier> "=" "input"

<statements> ::=
      <def_fun>
      | <call_fun>
      | <output>
      | <for_stmt>
      | <if_stmt>
      | <while_stmt>
      | <def_var>
      | <return>
      | <break>
      | <skip>
      | <input>
      | <expression>

<identifier> ::=
      /^(?=.*?[a-zA-Z_][a-zA-Z0-9_]*)((?!^program$|^endprogram$|^fun$|^endfun$|
^if$|^else$|^endif$|^while$|^endwhile$|^for$|^in$|^endfor$|^true$|^false$|
^void$|^and$|^or$|^not$|^return$|^break$|^skip$|^input$|^int$|^float$|^array$|
^string$|^bool$|^output$|"|'|^\/\/.*\n+).)*$/

<variable> ::=
      <identifier>
```

```
<fun_type> ::=
      <type>
      | <void>

<array_type> ::=
      "int" "array"
      | "float" "array"
      | "string" "array"
      | "bool" "array"

<type> ::=
      "int"
      | "float"
      | "string"
      | "bool"

<output> ::=
      "output" <output_exprs>

<output_exprs> ::=
      <output_exprs> "," <expression>
      | <output_exprs> "," <string>
      | <output_exprs> "," <call_fun>
      | <output_exprs> "," <array>
      | <call_fun>
      | <expression>
      | <string>
      | <array>
```

4.2 Programkoden

4.2.1 RDParser

```
#!/usr/bin/env ruby

# 2010-02-11 New version of this file for the 2010 instance of TDP007
#   which handles false return values during parsing, and has an easy way
#   of turning on and off debug messages.

require 'logger'

@@log = false

class Rule

  # A rule is created through the rule method of the Parser class, like this:
  #   rule :term do
  #     match(:term, '*', :dice) {|a, _, b| a * b }
  #     match(:term, '/', :dice) {|a, _, b| a / b }
  #     match(:dice)
  #   end

  Match = Struct.new :pattern, :block

  def initialize(name, parser)
    @logger = parser.logger
    # The name of the expressions this rule matches
    @name = name
    # We need the parser to recursively parse sub-expressions occurring
    # within the pattern of the match objects associated with this rule
    @parser = parser
    @matches = []
    # Left-recursive matches
    @lrmatches = []
  end

  # Add a matching expression to this rule, as in this example:
  #   match(:term, '*', :dice) {|a, _, b| a * b }
  # The arguments to 'match' describe the constituents of this expression.
  def match(*pattern, &block)
    match = Match.new(pattern, block)
    # If the pattern is left-recursive, then add it to the left-recursive set
    if pattern[0] == @name
      pattern.shift
      @lrmatches << match
    else
      @matches << match
    end
  end

  def parse
    # Try non-left-recursive matches first, to avoid infinite recursion
    match_result = try_matches(@matches)
    return nil if match_result.nil?
    loop do
      result = try_matches(@lrmatches, match_result)
      return match_result if result.nil?
      match_result = result
    end
  end
```

```
  private

  # Try out all matching patterns of this rule
  def try_matches(matches, pre_result = nil)
    match_result = nil
    # Begin at the current position in the input string of the parser
    start = @parser.pos
    matches.each do |match|
      # pre_result is a previously available result from evaluating expressions
      result = pre_result ? [pre_result] : []

      # We iterate through the parts of the pattern, which may be e.g.
      #   [:expr,'*',:term]
      match.pattern.each_with_index do |token,index|

        # If this "token" is a compound term, add the result of
        # parsing it to the "result" array
        if @parser.rules[token]
          result << @parser.rules[token].parse
          if result.last.nil?
            result = nil
            break
          end
          if @@log
                    @logger.debug("Matched  '#{@name}  =  #{match.pattern[index..-
1].inspect}'")
          end
        else
          # Otherwise, we consume the token as part of applying this rule
          nt = @parser.expect(token)
          if nt
            result << nt
            if @lrmatches.include?(match.pattern) then
              pattern = [@name]+match.pattern
            else
              pattern = match.pattern
            end
            if @@log
               @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <=
#{pattern.inspect}'")
            end
          else
            result = nil
            break
          end
        end
      end
      if result
        if match.block
          match_result = match.block.call(*result)
        else
          match_result = result[0]
        end
        if @@log
                            @logger.debug("Matched   '#{@name}'   and   generated
'#{match_result.inspect}'")          unless          match_result.nil?          #
@parser.string[start..@parser.pos-1]
        end
        break
      else
        # If this rule did not match the current token list, move
        # back to the scan position of the last match
        @parser.pos = start
```

```
      end
    end

    return match_result
  end
end

class Parser

  attr_accessor :pos
  attr_reader :rules, :string, :logger

  class ParseError < RuntimeError
  end

  def initialize(language_name, &block)
    @logger = Logger.new(STDOUT)
    @lex_tokens = []
    @rules = {}
    @start = nil
    @language_name = language_name
    instance_eval(&block)
  end

  # Tokenize the string into small pieces
  def tokenize(string)
    @tokens = []
    @string = string.clone
    until string.empty?
      # Unless any of the valid tokens of our language are the prefix of
      # 'string', we fail with an exception
       raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |
tok|
        match = tok.pattern.match(string)
            # The regular expression of a token has matched the beginning of
'string'
        if match
          if @@log
            @logger.debug("Token #{match[0]} consumed")
          end
          # Also, evaluate this expression by using the block
          # associated with the token
          @tokens << tok.block.call(match.to_s) if tok.block
          # consume the match and proceed with the rest of the string
          string = match.post_match
          true
        else
          # this token pattern did not match, try the next
          false
        end # if
      end # raise
    end # until
  end

  def parse(string)
    # First, split the string according to the "token" instructions given.
    # Afterwards @tokens contains all tokens that are to be parsed.
    tokenize(string)

    # These variables are used to match if the total number of tokens
    # are consumed by the parser
    @pos = 0
    @max_pos = 0
    @expected = []
```

```
    # Parse (and evaluate) the tokens received
    result = @start.parse
    # If there are unparsed extra tokens, signal error
    if @pos != @tokens.size
      raise ParseError, "Parse error. expected: '#{@expected.join(', ')}', found
'#{@tokens[@max_pos]}'"
    end
    return result
  end

  def next_token
    @pos += 1
    return @tokens[@pos - 1]
  end

  # Return the next token in the queue
  def expect(tok)
    t = next_token
    if @pos - 1 > @max_pos
      @max_pos = @pos - 1
      @expected = []
    end
    return t if tok === t
    @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
    return nil
  end

  def to_s
    "Parser for #{@language_name}"
  end

  private

  LexToken = Struct.new(:pattern, :block)

  def token(pattern, &block)
    @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
  end

  def start(name, &block)
    rule(name, &block)
    @start = @rules[name]
  end

  def rule(name,&block)
    @current_rule = Rule.new(name, self)
    @rules[name] = @current_rule
    instance_eval &block
    @current_rule = nil
  end

  def match(*pattern, &block)
    @current_rule.send(:match, *pattern, &block)
  end

end
```

4.2.2 Indenterings-check

```
#!/usr/bin/env ruby

def count_tabs(str,tab)
  tabs = 0
  left = 0
  for right in 0..str.size-1
    if(str.slice(left..right) == tab)
      tabs += 1
      left = right+1
    end
  end
  return tabs
end

def tab_test(str)
  pass = true
  add = ["program","for","while","if","fun"]
  sub = ["endprogram","endfor","endwhile","endif","endfun"]
  file = File.new(str, "r")
  wanted = 0
  tab = 0

  while(line = file.gets)
    if(/^[ \t]+\w+/ =~ line)
      tab = line.match(/(\s+)\w+/)[1]
      if(line.match(/\s+(\w+)/)[1] == "program")
        wanted = 1
      end
      break
    end
  end
  file.close

  file = File.new(str, "r")
  row = 0
  while(line = file.gets)
    row += 1

    if(/^\s+$/ =~ line)
      next
    end

    if(/^\s*\/\/.*/ =~ line)
      next
    end

    key = line.match(/#{tab}*(\/*\w+)/)[1]
    tabs = line.match(/(#{tab}*)\w+/)[1]
    found = 0

    if(key.slice(0,2) == "//")
      next
    end

    if(sub.include?(key))
      wanted -= 1
    end

    if(key != "else")
      found = count_tabs(tabs,tab)
```

```
    else
      found = count_tabs(tabs,tab)+1
    end

    if(found != wanted)
          raise "TTParser:<Line #{row}> Expected #{wanted} indent(s), found
#{found}!"
      pass = false
      break
    end

    if(add.include?(key))
      wanted += 1
    end

    if(key == "endprogram")
      pass = true
      break
    end

  end
  file.close
  return pass
end
```

4.2.3 TTParser

```
#!/usr/bin/env ruby

require 'rdparser'
require 'tabtest'

#############################################################################
#
# This part defines the TabTab gates
#
#############################################################################

@@stop = ["**return**","**break**","**skip**"]
@@type_convert = {
  Fixnum=>"int",
  Float=>"float",
  String=>"string",
  Array=>"array",
  TrueClass=>"bool",
  FalseClass=>"bool"}

def check_scope(name)
  value = [false,nil]
  index = @@scope.size
  while index > 0
    index -= 1
    if @@scope[index].has_key?(name)
      value = [true,index]
      return value
    end
  end
  return value
end


def array_check(type,value)
  right = true
  for element in value
    if @@type_convert[element.class] != type
      right = false
    end
  end
  return right
end


def type_check(name,value,type,is_array)
  set = false
  value_type = @@type_convert[value.class]
  if is_array
    if value_type == "array"
      @@scope[@@current_scope][name] = ["array", value, type]
      set = true
    end
  else
    if value_type == type
      @@scope[@@current_scope][name] = [type, value]
      set = true
    end
  end
```

```
  if not set
    raise "TTParser: Variable '#{name}' of type '#{type}' cannot be set to value
'#{value}'."
  end
end


class Program

  def initialize(stmts)
    @stmts = stmts
    eval()
  end

  def eval
    @stmts.eval()
  end
end


class Statements

  def initialize(stmt,stmts=nil)
    @stmt = stmt
    @stmts = stmts
  end

  def eval
    if(@stmts != nil)
      var = @stmts.eval()
      if var.class == Array
        if @@stop.include?(var[0])
          return var
        end
      end
    end
    var = @stmt.eval()
    if var.class == Array
      if @@stop.include?(var[0])
        return var
      end
    end
  end
end


class Statement

  def initialize(stmt)
    @stmt = stmt
  end

  def eval
    var = @stmt.eval()
    if var.class == Array
      if @@stop.include?(var[0])
        return var
      end
    end
  end
end


class Return
```

```
  def initialize(value)
    @value = value
  end

  def eval
    return ["**return**",@value]
  end
end


class Break

  def initialize
  end

  def eval
    return ["**break**"]
  end
end


class Skip

  def initialize
  end

  def eval
    return ["**skip**"]
  end
end


class Comment

  def initialize(comment)
    @comment = comment
  end

  def eval
    #Debug info?
    nil
  end
end


class Def_fun

  def initialize(type, name, stmts, paras=nil, is_array=false)
    @type = type
    @name = name
    @stmts = stmts
    @paras = paras
    @is_array = is_array
  end

  def eval
    if @is_array
          @@scope[@@current_scope][@name.eval()]  =  [Type.new("array"),  @paras,
@stmts, @type]
    else
      @@scope[@@current_scope][@name.eval()] = [@type, @paras, @stmts]
    end
  end
```

```
end


class Call_fun

  def initialize(name, args=nil)
    @name = name
    @args = args
  end

  def eval
    if check_scope(@name.eval())[0]
      has_returnval = false
      index = check_scope(@name.eval())[1]
      @type = @@scope[index][@name.eval()][0]
      stmts = @@scope[index][@name.eval()][2] #Retrieve function statements
      @@scope << {}
      @@current_scope += 1
      if @args != nil
        #Create variables if arguments are included
        arguments = @args.eval()
        parameters = @@scope[index][@name.eval()][1].eval()

        for index in (0...arguments.size)
          value = arguments[index]
          name = parameters[index][1]
          type = parameters[index][0]
          if @@type_convert[value.class] == "array"
            arr_type = parameters[2]
            type_check(name,value,arr_type,true)
          else
            type_check(name,value,type,false)
          end
        end
      end
      var = stmts.eval() #Run function statements
      if var.class == Array
        if var[0] == "**return**"
          has_returnval = true
        end
        if @@stop.include?(var[0]) and var[0] != "**return**"
          return var
        end
      end
      if has_returnval
        index = check_scope(@name.eval())[1]
        rtrn_value = var[1].eval()
        if @@type_convert[rtrn_value.class] == @type.eval()
                  if  @type.eval()  ==  "array"  and  array_check(@@scope[index]
[@name.eval()][3].eval(), rtrn_value)
            return rtrn_value
          else
            return rtrn_value
          end
        else
           raise "TTParser: Expected return value of type '#{@type.eval()}' from
function '#{@name.eval()}', found '#{@@type_convert[rtrn_value.class]}'!"
        end
      else
        if @type.eval() != "void"
           raise "TTParser: Expected return value of type '#{@type.eval()}' from
function '#{@name.eval()}', found none!"
        else
          return nil
```

```
        end
      end
      @@current_scope -= 1
      @@scope.pop()
      nil
    else
       raise "TTParser: Invalid function call. Function '#{@name.eval()}' does
not exist!"
    end
  end
end


class Parameters

  def initialize(para, paras=nil)
    @para = para
    @paras = paras
  end

  def eval
    if @paras != nil
      return @paras.eval() + [@para.eval()]
    else
      return [@para.eval()]
    end
  end
end


class Parameter

  def initialize(type, name, is_array=false)
    @type = type
    @name = name
    @is_array = is_array
  end

  def eval
    if @is_array
      return ["array", @name.eval(), @type.eval()]
    else
      return [@type.eval(),@name.eval()]
    end
  end
end


class Arguments

  def initialize(arg, args=nil)
    @arg = arg
    @args = args
  end

  def eval
    if @args != nil
      return @args.eval() + @arg.eval()
    else
      return @arg.eval()
    end
  end
end
```

```
class Argument

  def initialize(value)
    @value = value
  end

  def eval
     return [@value.eval()]
  end
end


class Output

  def initialize(exprs)
    @exprs = exprs
  end

  def eval
    for expr in @exprs.eval()
      if expr.eval() != nil
        puts expr.eval()
      end
    end
  end
end


class Input

  def initialize(var,typed=false)
    @var = var
    @typed = typed
  end

  def eval
    if @typed
      value = gets
      @@scope[@@current_scope][@var.eval()] = ["string", value[0...-1]]
    else
      if check_scope(@var.eval())[0]
        value = gets
        index = check_scope(@var.eval())[1]
        type = @@scope[index][@var.eval()][0]
        if type == "string"
          @@scope[index][@var.eval()][1] = value[0...-1]
        else
          raise "TTParser: Variable '#{@var.eval()}' is not of type string."
        end
      else
        raise "TTParser: Variable '#{@var.eval()}' is not defined."
      end
    end
  end
end


class For_stmt

  def initialize(name,container,stmts)
    @name = name
    @container = container
    @stmts = stmts
```

```
  end

  def get_element(cont,i)
    if cont.class == Array
      return cont[i..i][0]
    else
      return cont[i..i]
    end
  end

  def eval
    @@scope << {}
    @@current_scope += 1
    cont = @container.eval()
    @type = @@type_convert[cont[0].class]
    for i in (0...cont.size)
      @@scope[@@current_scope][@name.eval()] = [@type, get_element(cont,i)]
      var = @stmts.eval()
      if var.class == Array
        if var[0] == "**break**"
          break
        end
        if var[0] == "**skip**"
          next
        end
           if @@stop.include?(var[0]) and var[0] != "**break**" and var[0] !=
"**skip**"
          return var
        end
      end
    end
    @@current_scope -= 1
    @@scope.pop()
    nil
  end
end


class While_stmt

  def initialize(expr,stmts)
    @expr = expr
    @stmts = stmts
  end

  def eval
    @@scope << {}
    @@current_scope += 1
    while @expr.eval() do
      var = @stmts.eval()
      if var.class == Array
        if var[0] == "**break**"
          break
        end
        if var[0] == "**skip**"
          next
        end
           if @@stop.include?(var[0]) and var[0] != "**break**" and var[0] !=
"**skip**"
          return var
        end
      end
    end
    @@current_scope -= 1
```

```
    @@scope.pop()
    nil
  end
end


class If_stmt

  def initialize(expr,stmts,stmts_else=nil)
    @expr = expr
    @stmts = stmts
    @stmts_else = stmts_else
  end

  def eval
    if @stmts_else == nil
      if @expr.eval()
        @stmts.eval()
      end
    else
      if @expr.eval()
        @stmts.eval()
      else
        @stmts_else.eval()
      end
    end
  end
end


class Def_var

  def initialize(name, value, type=nil, is_array=false)
    @type = type
    @name = name
    @value = value
    @is_array = is_array
  end

  def eval
    if @type != nil
      if @is_array
        if @value.check_types(@type.eval())
          type_check(@name.eval(),@value.eval(),@type.eval(),@is_array)
        end
      else
        if @@types.include?(@value.eval().class)
          type_check(@name.eval(),@value.eval(),@type.eval(),@is_array)
        end
      end
    else
      if check_scope(@name.eval())[0]
        index = check_scope(@name.eval())[1]
        value = @value.eval()
        type = @@scope[index][@name.eval()][0]
        if type == @@type_convert[value.class]
          if type == "array"
            array_type = @@scope[index][@name.eval()][2]
            if array_check(array_type, value)
              @@scope[index][@name.eval()][1] = value
            else
                    raise "TTParser: Element(s) in '#{value}' is not of type
'#{array_type}'."
            end
```

```
          else
            @@scope[index][@name.eval()][1] = value
          end
        else
           raise "TTParser: Variable '#{@name.eval()}' of type '#{type}' cannot
be set to value '#{value}'."
        end
      else
        raise "TTParser: Variable '#{@name.eval()}' is not defined."
      end
    end
  end
end


class Output_exprs

  def initialize(expr,exprs=nil)
    @expr = expr
    @exprs = exprs
  end

  def eval
    if @exprs != nil
      return @exprs.eval() + [@expr]
    end
    return [@expr]
  end
end


class Expression

  def initialize(expr)
    @expr = expr
  end

  def eval
    @expr.eval()
  end
end


class Comp_expr

  def initialize(expr,op=nil,expr2=nil)
    @expr = expr
    @op = op
    @expr2 = expr2
  end

  def eval
    if @op == nil or @expr2 == nil
      @expr.eval()
    else
      operator = @op.eval()
      case operator
        when "<" then @expr.eval() < @expr2.eval()
        when ">" then @expr.eval() > @expr2.eval()
        when "==" then @expr.eval() == @expr2.eval()
        when "<=" then @expr.eval() <= @expr2.eval()
        when ">=" then @expr.eval() >= @expr2.eval()
        when "!=" then @expr.eval() != @expr2.eval()
      end
```

```
    end
  end
end


class Comp_operator

  def initialize(value)
    @value = value
  end

  def eval
    @value
  end
end


class Calc_expr

  def initialize(expr,op=nil,expr2=nil)
    @expr = expr
    @op = op
    @expr2 = expr2
  end

  def eval
    if @op == nil or @expr2 == nil
      @expr.eval()
    else
      operator = @op
      case operator
        when "+" then @expr.eval() + @expr2.eval()
        when "-" then @expr.eval() - @expr2.eval()
        when "*" then @expr.eval() * @expr2.eval()
        when "/" then @expr.eval() / @expr2.eval()
      end
    end
  end
end


class Neg_expr

  def initialize(expr,op=nil)
    @expr = expr
    @op = op
  end

  def eval
    if @op == "-"
      return -(@expr.eval())
    else
      return @expr.eval()
    end
  end
end


class Log_expr

  def initialize(expr,op=nil,expr2=nil)
    @expr = expr
    @op = op
    @expr2 = expr2
```

```
  end

  def eval
    if @op == nil or @expr2 == nil
      @expr.eval()
    else
      operator = @op
      case operator
        when "and" then @expr.eval() and @expr2.eval()
        when "or" then @expr.eval() or @expr2.eval()
      end
    end
  end
end


class Not_expr

  def initialize(expr,op=nil)
    @expr = expr
    @op = op
  end

  def eval
    if @op == "not"
      return (not @expr.eval())
    else
      return @expr.eval()
    end
  end
end


class Type

  def initialize(value)
    @value = value
  end

  def eval
    @value
  end
end


class Int_type

  def initialize(value)
    @value = value
  end

  def eval
    @value
  end
end


class Float_type

  def initialize(value)
    @value = value
  end

  def eval
```

```
    @value
  end
end


class String_type

  def initialize(value)
    @value = value
  end

  def eval
    @value.slice(1,@value.size-2)
  end
end


class Bool_type

  def initialize(value)
    @value = value
  end

  def eval
    @value
  end
end


class Array_elems

  def initialize(arg, args=nil)
    @arg = arg
    @args = args
  end

  def eval
    if @args != nil
      return @args.eval() + @arg.eval()
    else
      return @arg.eval()
    end
  end
end


class Array_elem

  def initialize(value)
    @value = value
  end

  def eval
      return [@value.eval()]
  end
end


class Array_type

  def initialize(elements=nil)
    @elements = elements
  end
```

```
  def eval
    if @elements != nil
      return @elements.eval()
    else
      return []
    end
  end

  def check_types(type)
    right = true
    for element in @elements.eval()
      if @@type_convert[element.class] != type
        right = false
                    raise "TTParser: Element '#{element}' is of type
'#{@@type_convert[element.class]}', expected '#{type}'!"
      end
    end
    return right
  end
end


class Identifier

  def initialize(value)
    @value = value
  end

  def eval
    @value
  end
end


class Variable

  def initialize(value)
    @value = value
  end

  def eval
    var = @value.eval()
    if check_scope(var)[0]
      return @@scope[check_scope(var)[1]][var][1]
    else
      raise "TTParser: No variable named '#{var}'."
    end
  end
end


class Container

  def initialize(cont)
    @cont = cont
  end

  def eval
    value = @cont.eval()
    type = @@type_convert[value.class]
    if type == "array" or type == "string"
      return value
    else
      raise "TTParser: Variable is not a valid container."
```

```
    end
  end
end


################################################################################
#
# This part defines the TabTab language
#
################################################################################
@@scope = []
@@scope << {}
@@current_scope = 0
@@types = [Fixnum,Float,String,Array,TrueClass,FalseClass]

class TabTab

  def initialize
    @TTParser = Parser.new("TabTab") do
      #Newline
      token(/\s*\n+/) {"\n"}
      #Comment
      token(/\/\/.*\s*\n+/) {|m| m}
      #Remove whitespace
      token(/\s+/)
      token(/ /)
      #Float
      token(/\d+\.\d+/) {|m| m.to_f}
      #Int
      token(/\d+/) {|m| m.to_i}
      #String
      token(/"[^"]*[^\\]"/) {|m| m} #crash on "\\"
      token(/'[^']*[^\\]'/) {|m| m}
      #Parenthesis
      token(/[\(\)]/) {|m| m}
      #Calc Operators
      token(/\+/) {|m| m}
      token(/-/) {|m| m}
      token(/\*/) {|m| m}
      token(/\//) {|m| m}
      #Comp Operators
      token(/<=/) {|m| m}
      token(/>=/) {|m| m}
      token(/==/) {|m| m}
      token(/!=/) {|m| m}
      token(/</) {|m| m}
      token(/>/) {|m| m}
      #Comma
      token(/,/) {|m| m}
      #Assignment Operator
      token(/=/) {|m| m}
      #Array
      token(/\[/) {|m| m}
      token(/\]/) {|m| m}
      #Word
      token(/[a-zA-Z_][a-zA-Z0-9_]*/) {|m| m}
      #Everything else
      token(/.+/) {|m| m}

      start :program do
         match("program", "\n", :statements, "endprogram", "\n"){|_,_,stmts,_,_|
Program.new(stmts)
        nil}
```

```
      end

      rule :def_fun do
            match("fun", :fun_type, :identifier, "(", :parameters, ")", "\n",
:statements,             "endfun")            {|_,type,name,_,args,_,_,stmts,_|
Def_fun.new(type,name,stmts,args)}
           match("fun", :fun_type, :identifier, "(", ")", "\n", :statements,
"endfun") {|_,type,name,_,_,_,stmts,_| Def_fun.new(type,name,stmts)}
          match("fun", :array_type, :identifier, "(", :parameters, ")", "\n",
:statements,             "endfun")            {|_,type,name,_,args,_,_,stmts,_|
Def_fun.new(type,name,stmts,args,true)}
          match("fun", :array_type, :identifier, "(", ")", "\n", :statements,
"endfun") {|_,type,name,_,_,_,stmts,_| Def_fun.new(type,name,stmts,nil,true)}
      end

      rule :call_fun do
               match(:identifier,  "(",    :arguments,  ")")  {|name,_,args,_|
Call_fun.new(name,args)}
        match(:identifier, "(", ")") {|name,_,_| Call_fun.new(name)}
      end

      rule :for_stmt do
         match("for", "(",:identifier, "in", :container, ")", "\n", :statements,
"endfor")                                  {|_,_,name,_,container,_,_,stmts,_|
For_stmt.new(name,container,stmts)}
      end

      rule :while_stmt do
         match("while", "(", :expression, ")", "\n", :statements, "endwhile") {|
_,_,expr,_,_,stmt,_| While_stmt.new(expr,stmt)}
      end

      rule :if_stmt do
            match("if", "(", :expression, ")", "\n", :statements, "endif") {|
_,_,expr,_,_,stmt,_| If_stmt.new(expr,stmt)}
          match("if", "(", :expression, ")", "\n", :statements, "else", "\n",
:statements,             "endif")             {|_,_,expr,_,_,stmt,_,_,stmt2,_|
If_stmt.new(expr,stmt,stmt2)}
      end

      rule :comment do
        match(/\/\/.*\n/ ) {|value| Comment.new(value)}
      end

      rule :def_var do
             match(:type, :identifier, "=", :call_fun)  {|type,name,_,value|
Def_var.new(name,value,type)}
                      match(:identifier,   "=",   :call_fun)   {|name,_,value|
Def_var.new(name,value)}
            match(:type, :identifier, "=", :expression)  {|type,name,_,value|
Def_var.new(name,value,type)}
                     match(:identifier,   "=",   :expression)   {|name,_,value|
Def_var.new(name,value)}
               match(:type, :identifier, "=", :string)  {|type,name,_,value|
Def_var.new(name,value,type)}
                       match(:identifier,   "=",   :string)   {|name,_,value|
Def_var.new(name,value)}
           match(:array_type, :identifier, "=", :array)  {|type,name,_,value|
Def_var.new(name,value,type,true)}
                        match(:identifier,   "=",   :array)   {|name,_,value|
Def_var.new(name,value)}#(name,value,nil,array)?
      end

      rule :container do
```

```
        match(:string) {|cont| Container.new(cont)}
        match(:array) {|cont| Container.new(cont)}
        match(:variable) {|cont| Container.new(cont)}
      end

      rule :int do
        match(Integer) {|value| Int_type.new(value)}
      end

      rule :float do
        match(Float) {|value| Float_type.new(value)}
      end

      rule :array_elem do
        match(:string) {|value| Array_elem.new(value)}
        match(:expression) {|value| Array_elem.new(value)}
      end

      rule :array_elems do
                    match(:array_elems,   ",",   :array_elem)   {|values,_,value|
Array_elems.new(value, values)}
        match(:array_elem) {|value| Array_elems.new(value)}
      end

      rule :array do
        match("[", :array_elems, "]") {|_,elements,_| Array_type.new(elements)}
        match("[","]") {Array_type.new()}
      end

      rule :string do
        match(/".*"/) {|value| String_type.new(value)}
        match(/'.*'/) {|value| String_type.new(value)}
      end

      rule :bool do
        match(:true) {|value| Bool_type.new(value)}
        match(:false) {|value| Bool_type.new(value)}
      end

      rule :true do
        match("true") {true}
      end

      rule :false do
        match("false") {false}
      end

      rule :parameter do
        match(:type, :identifier) {|type,name| Parameter.new(type,name)}
                              match(:array_type,    :identifier)    {|type,name|
Parameter.new(type,name,true)}
      end

      rule :parameters do
        match(:parameters, ",", :parameter) {|paras,_,para| Parameters.new(para,
paras)}
        match(:parameter) {|para| Parameters.new(para)}
      end

      rule :argument do
        match(:variable) {|value| Argument.new(value)}
        match(:expression) {|value| Argument.new(value)}
        match(:string) {|value| Argument.new(value)}
        match(:array) {|value| Argument.new(value)}
```

```
      end

      rule :arguments do
            match(:arguments, ",", :argument) {|args,_,arg| Arguments.new(arg,
args)}
        match(:argument) {|arg| Arguments.new(arg)}
      end

      rule :expression do
        match(:or_expr) {|expr| Expression.new(expr)}
      end

      rule :para do
        match("(",:add_calc_expr,")") {|_,expr,_| Calc_expr.new(expr)}
      end

      rule :num do
        match(:para) {|expr| Calc_expr.new(expr)}
        match(:int) {|expr| Calc_expr.new(expr)}
        match(:float) {|expr| Calc_expr.new(expr)}
        match(:variable) {|expr| Calc_expr.new(expr)}
      end

      rule :neg_expr do
        match(:num) {|expr| Neg_expr.new(expr)}
        match("-", :num) {|op,expr| Neg_expr.new(expr,op)}
      end

      rule :multi_calc_expr do
        match(:neg_expr) {|expr| Calc_expr.new(expr)}
              match(:multi_calc_expr, "*", :multi_calc_expr) {|expr,op,expr2|
Calc_expr.new(expr,op,expr2)}
              match(:multi_calc_expr, "/", :multi_calc_expr) {|expr,op,expr2|
Calc_expr.new(expr,op,expr2)}
      end

      rule :add_calc_expr do
        match(:multi_calc_expr) {|expr| Calc_expr.new(expr)}
               match(:add_calc_expr, "+", :multi_calc_expr) {|expr,op,expr2|
Calc_expr.new(expr,op,expr2)}
               match(:add_calc_expr, "-", :multi_calc_expr) {|expr,op,expr2|
Calc_expr.new(expr,op,expr2)}
      end

      rule :calc_expr do
        match(:add_calc_expr)
      end

      rule :comp_expr do
             match(:calc_expr, :comp_operator, :calc_expr) {|expr,op,expr2|
Comp_expr.new(expr,op,expr2)}
        match(:calc_expr) {|expr| Comp_expr.new(expr)}
        match(:bool) {|expr| Comp_expr.new(expr)}
        match(:variable) {|expr| Comp_expr.new(expr)}
      end

      rule :comp_operator do
        match("<") {|op| Comp_operator.new(op)}
        match(">") {|op| Comp_operator.new(op)}
        match("==") {|op| Comp_operator.new(op)}
        match("!=") {|op| Comp_operator.new(op)}
        match("<=") {|op| Comp_operator.new(op)}
        match(">=") {|op| Comp_operator.new(op)}
      end
```

```
      rule :not_expr do
        match(:comp_expr) {|expr| Not_expr.new(expr)}
        match("not", :not_expr) {|op,expr| Not_expr.new(expr,op)}
      end

      rule :and_expr do
        match(:not_expr) {|expr| Log_expr.new(expr)}
                              match(:and_expr,"and",:not_expr)    {|expr,op,expr2|
Log_expr.new(expr,op,expr2)}
      end

      rule :or_expr do
        match(:and_expr) {|expr| Log_expr.new(expr)}
                               match(:or_expr,"or",:and_expr)    {|expr,op,expr2|
Log_expr.new(expr,op,expr2)}
      end

      rule :return do
        match("return", :expression) {|_,expr| Return.new(expr)}
        match("return", :string) {|_,str| Return.new(str)}
        match("return", :call_fun) {|_,call| Return.new(call)}
        match("return", :array) {|_,arr| Return.new(arr)}
      end

      rule :break do
        match("break") {Break.new()}
      end

      rule :skip do
        match("skip") {Skip.new()}
      end

      rule :input do
                    match("string",   :identifier,   "=",   "input")   {|_,var,_,_|
Input.new(var,true)}
        match(:identifier, "=", "input") {|var,_,_| Input.new(var)}
      end

      rule :statements do
                               match(:statements,    :lined_stmt)    {|stmts,stmt|
Statements.new(stmt,stmts)}
        match(:lined_stmt)
      end

      rule :lined_stmt do
        match(:statement, "\n")
        match(:statement, :comment)
        match(:comment)
      end

      rule :statement do
        match(:def_fun) {|stmt| Statement.new(stmt)}
        match(:call_fun) {|stmt| Statement.new(stmt)}
        match(:output) {|stmt| Statement.new(stmt)}
        match(:for_stmt) {|stmt| Statement.new(stmt)}
        match(:if_stmt) {|stmt| Statement.new(stmt)}
        match(:while_stmt) {|stmt| Statement.new(stmt)}
        match(:def_var) {|stmt| Statement.new(stmt)}
        match(:return) {|stmt| Statement.new(stmt)}
        match(:break) {|stmt| Statement.new(stmt)}
        match(:skip) {|stmt| Statement.new(stmt)}
        match(:input) {|stmt| Statement.new(stmt)}
        match(:expression) {|stmt| Statement.new(stmt)}
```

```
      end

      rule :identifier do
          match(/^(?=.*?[a-zA-Z_][a-zA-Z0-9_]*)((?!^program$|^endprogram$|^fun$|
^endfun$|^if$|^else$|^endif$|^while$|^endwhile$|^for$|^in$|^endfor$|^true$|
^false$|^void$|^and$|^or$|^not$|^return$|^break$|^skip$|^input$|^int$|^float$|
^array$|^string$|^bool$|^output$|"|'|^\/\/.*\n+).)*$/)                    {|value|
Identifier.new(value)}
      end

      rule :variable do
        match(:identifier) {|value| Variable.new(value)}
      end

      rule :fun_type do
        match(:type)
        match("void") {|type| Type.new(type)}
      end

      rule :array_type do
        match("int","array") {|type,_| Type.new(type)}
        match("float","array") {|type,_| Type.new(type)}
        match("string","array") {|type,_| Type.new(type)}
        match("bool","array") {|type,_| Type.new(type)}
      end

      rule :type do
        match("int") {|type| Type.new(type)}
        match("float") {|type| Type.new(type)}
        match("string") {|type| Type.new(type)}
        match("bool") {|type| Type.new(type)}
      end


      rule :output do
        match("output", :output_exprs) {|_,expr| Output.new(expr)}
      end

      rule :output_exprs do
                        match(:output_exprs,   ",",   :expression)   {|exprs,_,expr|
Output_exprs.new(expr,exprs)}
                          match(:output_exprs,   ",",   :string)   {|exprs,_,str|
Output_exprs.new(str,exprs)}
                         match(:output_exprs,   ",",   :call_fun)   {|exprs,_,call|
Output_exprs.new(call,exprs)}
                           match(:output_exprs,   ",",   :array)   {|exprs,_,arr|
Output_exprs.new(arr,exprs)}
        match(:call_fun) {|call| Output_exprs.new(call)}
        match(:expression) {|expr| Output_exprs.new(expr)}
        match(:string) {|str| Output_exprs.new(str)}
        match(:array) {|arr| Output_exprs.new(arr)}
      end

    end
  end


  def run(new_file=nil)
    if(new_file.split(".")[-1] == "tt")
      if(tab_test(new_file))
        file = File.new(new_file, "r")
        str = ""
        while(line = file.gets)
          str += line
```

```
          if((/endprogram *\n*/ =~ line) != nil)
            break
          end
        end
        file.close
        puts "#{@TTParser.parse str}"
      end
    else
      raise "TTParser: Invalid file type, must be of type '.tt'!"
    end
  end

  def log(state = false)
    if state
      @TTParser.logger.level = Logger::DEBUG
    else
      @TTParser.logger.level = Logger::WARN
    end
  end
end
```