Mathematica като език за програмиране

Изходен код (Source Code), Интерпретирани и компилирани езици

Нека започнем с дефинирането на няколко термина. Вече в началото въведохме няколко термина: вече знаете с какви видове данни може да боравите, защото операторите имат предимство при обработката на изразите и защо това е от значение и какви са булевите стойности и булевите оператори

Инструкциите, които пишем на компютрите, на който и да е компютърен език основно се наричат source code (изходен код). Всички инструкции в *Mathematica,* команди или изрази, които се обработват във входните по-горе са примери за изходен код. Микропроцесорите обаче, като този в компютрите, на които работите, не разбират този изходен код. Те “разбират” само машинния код. Следователно програмистите са изправени пред проблема как този код да бъде преобразуван в машинен за да може да бъде обработен от процесора. В зависимост от конкретния език това може да стане по един от следните два начина. Или програмата преобразува изходния код в машинен всеки път, когато искаме да бъде изпълнен от компютъра или предварително е преобразуван. В първия случай се казва, че изходния код се интерпретира, а във втория - компилира.

Кой вариант е по-добър решава програмиста в зависимост от това кое за него е приоритет - лекотата на програмиране или бързото изпълнение (стартиране). Тъй като процесът на непосредствено преобразуване преди изпълнение отнема достатъчно дълго време, интерпретираните езици са доста по-бавни от компилираните. На практика всички програми, които са инсталирани на Вашия компютър са компилирани предварително, защото се очаква, че ще стартирате тези програми стотици или хиляди пъти и всеки път ще очаквате тези програми да се заредят (да стартират) възможно най-бързо. Представете си, че не искаме да стартираме нашите програми толкова често, а в замяна на това да си спестим изморителния процес на писане на кода, съставянето и управлението, откриването на грешки, пренаписването на кода, прекомпилирането и т.н. Да предположим, че вместо това искаме да коригираме кода и след това да стартираме програмата отново. В този случай интерпретатираните езици, като този на *Mathematica,* са за предпочитане. Забележете, че можете да управлявате всяка клетка с код по всяко време и тогава да я обработвате, като това става независимо от останалите клетки. При всяко обработване на клетката страва това преобразуване от изходен код в машинен.

Динамично програмиране (функция NSolve)

Mathematica притежава вградена команда "NSolve[]", която взима като аргумент уравнение и се опитва да намери числено решение (числени стойности на променливата, за които уравнението е вярно). Например:

```
In[1]:= NSolve[x^2==4,x]

Out[1]= {{x → -2.}, {x → 2.}}
```

```
In[2]:= NSolve[x^2==-4,x]
Out[2]= {{x → 0. - 2. ⅈ}, {x → 0. + 2. ⅈ}}
```

Първият аргумент на функцията "NSolve[]" е уравнението, съдържащо променливата, като отново обръщаме внимание, че знакът за равенство е записан двоен ("=="). Вторият аргумент е променливата. Забелязваме, че стойностите на променливата, които Mathematica намира като решение на уравнението са поставени в скоби, определящи списък, а тези списъци са част от по-голям списък. Самите стойности се изразяват по начин, който в Mathematica се нарича правило, където променливата е от лявата страна на стрелката ("->"), а променливата от дясно. Функцията "NSolve[]" може да се използва за решаване на много много уравнения, но понякога Mathematica се проваля в намирането на числено решенение. Направете сравнение решавайки изразите по-долу:

```
In[3]:= NSolve[Exp[x] == x+2, x]
NSolve::ifun : Inverse functions are being used by NSolve,
    so some solutions may not be found; use Reduce for complete solution information. >>
Out[3]= {{x → -1.84141}, {x → 1.14619}}

In[4]:= NSolve[Exp[x] == Abs[x+2], x]
NSolve::ifun : Inverse functions are being used by NSolve,
    so some solutions may not be found; use Reduce for complete solution information. >>
Out[4]= {{x → -1.84141}, {x → 1.14619}}

In[5]:= NSolve[Exp[x] == x + 2 Cos[x], x]
NSolve::nsmet : This system cannot be solved with the methods available to NSolve. >>
Out[5]= NSolve[ⅇ^x == x + 2 Cos[x], x]
```

Нека за момент да спрем с намирането на най-доброто решение, чрез функцията "NSolve[]". Да се концентрираме към динамичното програмиране, което означава процес на запазване на решенията намерени при всяко изпълнение на командата "NSolve[]". Ако намери решение, то може следващия път когато стартираме командата тя просто да направи проверка на предишния резултат и това да спести значително време за обработка отколкото ако се опита да реши уравнението от начало. С каква цел, обаче, да правим това? Тук отново въпросът опира до правенето на компромис. Компромис в това дали искаме да спестим оперативна памет и да олекотим програмата или целим по-бързо време за обработка. Ако искаме да не заема много памет, тогава принуждаваме *Mathematica* да намира решения на уравненията отново и отново. В този случай динамичното програмиране, както можете да си представите, се оказва лош избор, защото набора от уравнения е огромен и няма компютър с толкова много памет, че да запази решенията на всички уравнения. За решението на други проблеми обаче, динамичното програмиране може да се окаже много подходящ избор. Този пример показва друг компромис - пространство срещу време.

Прости и комплексни алгоритми (функция FindRoot)

В случаите, когато функцията "NSolve[]" не работи, *Mathematica* разполага с по-подходяща функция "FindRoot[]". "FindRoot[]" използва математически процес наречен метод на Нютон, който започва с оценка на корена на уравнението - стойност на променливата, при която

уравнението е вярно и се опитва постепенно да намери все по-точно и по-точно приближение. Например:

```
In[6]:= FindRoot[x^2==4,{x,1}]
Out[6]= {x → 2.}
```

Разбира се, съществува и друг корен (а именно x=-2), но започвайки от 1 програмата е достигнала до най-близкият и точен корен 2, при който уравнението е вярно. Нека опитаме функцията "FindRoot[]" за някои уравненията, които функцията "NSolve[]" не може да реши.

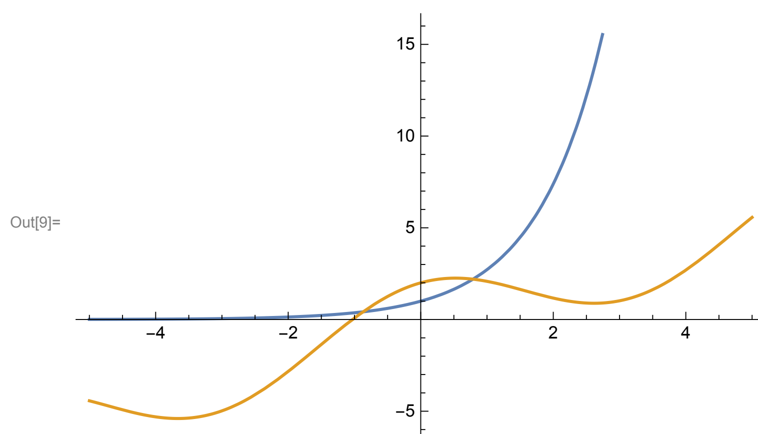
```
In[7]:= FindRoot[Exp[x]==x+2Cos[x],{x,1}]
Out[7]= {x → 0.787822}
```

```
In[8]:= FindRoot[Exp[x]==x+2Cos[x],{x,-1}]
Out[8]= {x → -0.870332}
```

Как може да предположим дали да започнем апроксимирането от 1 или -1? Един от начините е да начертаем графиките на двете функции "Exp[x]" и "x+2Cos[x]" и да видим къде приблизително те биха се пресекли.

```
In[9]:= Plot[{Exp[x], x + 2 Cos[x]},{x,-5,5}]
```

Out[9]=



Понякога *Mathematica* ни предупреждава, че може да пропусне решение, а понякога не. Нека отново разгледаме примерите по-горе

```
Exp[x] == x+2
```

и

```
Exp[x] == Abs[x+2]
```

и да се опитаме да ги решим:

```
In[10]:= NSolve[Exp[x] == x+2, x]
```

NSolve::ifun : Inverse functions are being used by NSolve,
so some solutions may not be found; use Reduce for complete solution information. >>

```
Out[10]= {{x → -1.84141}, {x → 1.14619}}
```

In[11]:=
```
NSolve[Exp[x] == Abs[x+2], x]
```

NSolve::ifun : Inverse functions are being used by NSolve,
so some solutions may not be found; use Reduce for complete solution information. >>

Out[11]=
```
{{x → -1.84141}, {x → 1.14619}}
```

Начертаваме и тяхните графики

In[12]:=
```
Plot[{Exp[x],x+2},{x,-5,5}];
```

In[13]:=
```
Plot[{Exp[x], Abs[x+2]}, {x,-4,2}, PlotRange->{0,4}];
```

Mathematica ни предупреждава, че може да пропусне решения на уравнението Exp[x] == x+2, но въпреки това намира всичките. От друга страна не ни предупреждава за уравнението Exp[x] == Abs[x+2], но изпуска всичките три решения.

Някои курсове за решаване на задачи препоръчват процеса на приближение към корен на функция (решението при f(x) = 0), наречен метод на Нютон. Методът на Нютон е разработен преди стотици години и се е утвърдил като добър метод за справяне с този проблем. Това е итеративен метод за последователно подобряване на точността. Идеята е да започнете от първоначална по-груба стойност за истинското решение и да замените “прогнозата си” с по-точна. Продължавате да търсите докато не постигнете желаната от Вас точност. Стъпката, която подобрява прогнозния корен е както следва:

Нека намерим число "x", което удовлетворява функцията "f(x)". Намираме първата производна "df/dx" на функцията. От предполагаемата стойност за “x” изваждаме отношението "f(x)/(df/dx)" и получаваме нов по-точен резултат. Повтаряме процедурата докато получим решение с желаната точност.

Методът на Нютон може да Ви се стори ненужно сложен. Някой може да предположи, че съществува по-прост метод за намиране на точка, където две функции се пресичат, използвайки следния аргумент: Преобразувайте уравнението така, че дясната страна да е нула. Сега започнете да замествате стойности в уравнението, започвайки от 1, докато не получите две стойности с противоположни знаци, които се повтарят от лявата страна на уравнението. Може да твърдим, че корена на уравнението е мужду тези две числа. След това започваме да проверяваме числата на равни интервали между тези две стойности (например през 1/10). Нека направим това с функциите по-горе. Ще определим функция “f[x], която ни дава лявата част "Exp[x]-(x+2Cos[x])==0".

In[14]:=
```
Clear[f];
f[x_] := Exp[x]-(x + 2 Cos[x])
```

In[16]:=
```
f[0.]
```

Out[16]=
```
-1.
```

In[17]:=
```
f[1.]
```

Out[17]=
```
0.637677
```

В този случай вече може да предположим, че корена на уравнението е между 0 и 1In this case, we already guess that a root lies between 0 and 1, and we explore further by examining "f[1.0]", "f[1.1]", "f[1.2]", "f[1.3]", and so on.

По-лесен ли е този метод? Вероятно не, но е много по-добър от метода на Нютон. Нека покажем предимството на този метод, като разгледаме следния пример с намирането на

приближено решение на уравнението (x – 0.5)^2 = 0.15. Нека опитаме метода на Нютон с вградената функция FindRoot, като започнем от начално предположение 0.

```
In[18]:= FindRoot[(x-.5)^2==.15,{x,0}]
Out[18]= {x → 0.112702}
```

В действителност уравнението има две решения между 0 и 1, както е показано на фигурата по-долу.

```
In[19]:= Plot[{(x-.5)^2, .15}, {x,0,1}];
```

Сега опитайте “по-лесния” метод.

```
In[20]:= Clear[f]
         f[x_] := (x-.5)^2 - .15;

In[22]:= f[0]
Out[22]= 0.1

In[23]:= f[1]
Out[23]= 0.1
```

Забележете, че макар уравнението (x - 0.5)^2 = 0.15 да има две решения между x=0 и x=1, “по-лесния” метод се проваля и не намира решения в посочения интервал. Това демонстрира третия компромис в компютърното програмиране. Понякога по-интуитивният метод се проваля, не защото не работи по принцип, а защото работи само в определени случаи. Докато всички програмисти биха искали да работят само с прости методи, те трябвна да балансират между желанието си за простота, коректност на получения резултата, времето, паметта и пространството на твърдия диск. В този курс ще бъдат представени някои задачи, които изискват малко по-сложни решения и трябва да предложите решение, което работи при всички случаи. Също така помага ако изпробвате решението си с много различни примери.