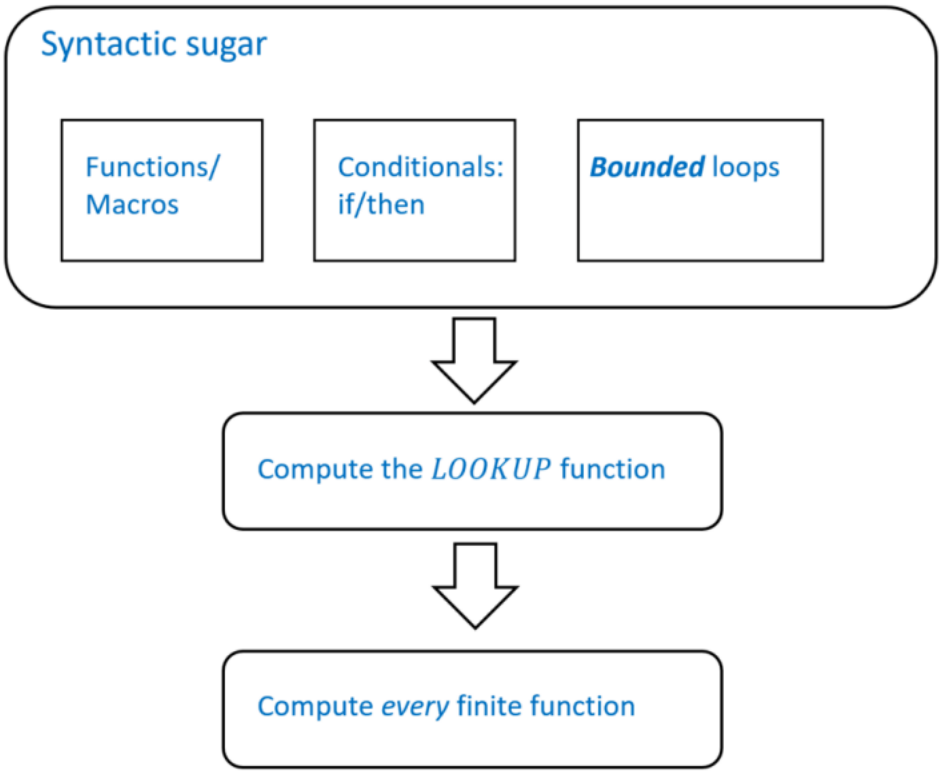
فصل ۴ بخش اول:

- **اهداف فصل:**

ارائه یک toolkit از «syntactic sugar» که بدست آوردن فیچر هایی مانند توابع تعریف شده توسط برنامه نویس و شرط ها را از NAND-CIRC ها نمایش دهیم.

و استفاده از این toolkit برای ساخت یک برنامه با NAND-CIRC برای محاسبه LOOKUP-FUNCTION و در نهایت نتیجه گیری از این پیاده سازی ها که هر فانکشن متنهی را می توان با NAND-CIRC پروگرم ها پیاده سازی کرد.(دو رویکرد برای اثبات این مورد ارائه می شود)



- **مفهوم syntactic sugar :**

پیاده سازی فیچر های پیچیده تر با استفاده از از فانکشنالیتی های موجود و و استفاده ازاین فیچر های ساخته شده برای ساختن فیچر های سطح بالا تر که این عمل که در واقع دستکاری مدل و مفهوم پایه یک زبان برنامه نویسی نیست و فقط بسط دادن توان موجود هست را "syntactic sugar" می گویند.(i.e. ما می توانیم یک زبان برنامه نویسی را با پیاده سازی ویژگی های پیشرفته از اجزاء ابتدایی آن گسترش دهیم.)

در کل هدف نمایش این است که با AON-CIRC (and-or-not) و یا NAND-CIRC ها هر فانکشن متنهی مثل شرط ها یا دیگر فانکشن های تعریف شده توسط کاربر را می توان پیاده سازی کرد و با یک جدول صحت نیز می توان عکس این روند را نشان داد بدین صورت که هر ورودی را به خروجی مربوطه خود مپ کنیم.

نمایش یک فانکشن متنهی : $f : \{0,1\}^n \to \{0,1\}^m$ ←

که ثابت می شود هر فانکشن متنهی با مدارات Boolean محاسبه می شود: این در واقع همان«universality» AND، OR، و NOT است. که همان معادل، NAND است.

- **4.1 چند مثال از فانکشن ها و توابع نوشته شده با syntactic sugar:**

از زبان برنامه نویسی NAND-CIRC و منطق زبان های برنامه نویسی straight-line استفاده می کنیم.

(یادآوری ـ فقط برای درک بهتر- دانستن آن ضروری نیست ـ : یک زبان برنامه نویسی straight-line یک مدل محاسباتی ساده با ویژگی های کلیدی زیر است:
Sequential execution: دستورالعمل ها به ترتیب خطی، یکی پس از دیگری، بدون هیچ گونه حلقه، شاخه یا پرش اجرا می شوند.
No recursion: زبان فراخوانی بازگشتی procedure را اجازه نمی دهد.
Variables and assignments: امکان تخصیص مقادیر به متغیرها و استفاده از آنها در محاسبات را فراهم می کند.

Acyclic structure: برنامه را می توان به عنوان یک گراف غیر چرخه جهت دار (DAG) نشان داد، که در آن گره ها عملیات را نشان می دهند و لبه ها جریان داده را نشان می دهند.
Deterministic output: برای یک ورودی معین، یک برنامه خط مستقیم همیشه همان خروجی را تولید می کند، زیرا هیچ عبارت شرطی یا حلقه ای وجود ندارد که بتواند مسیر اجرا را تغییر دهد.
Finite computation: برنامه پس از تعداد ثابتی از مراحل که با تعداد دستورات موجود در برنامه تعیین می شود، پایان می یابد.)

چرا با

برای اینکه ببینیم که با وجود ضعف و محدودیت های ظاهری، مدل های ساده مانند مدارهای Boolean یا NAND-CIRC در واقع بسیار قدرتمند هستند.(منظور استفاده از این ها به عنوان یک زبان برنامه نویسی است.)

*هایلایت مربوط به فصل قبل

هر نتیجه ای ما از تبدیلات syntactic sugar (استفاده از زبان برنامه نویسی NAND-CIRC برای ساخت انواع procedure ها و function ها) زیر بگیریم بدلیل تئوری زیر صادق است.

Theorem 3.19 — Equivalence between models of finite computation. For

every sufficiently large s, n, m and $f : \{0,1\}^n \to \{0,1\}^m$, the

following conditions are all **equivalent** to one another:

• f can be computed by a Boolean circuit (with ∧, ∨, ¬ gates) of at

most $O(s)$ gates.

• f can be computed by an AON-CIRC straight-line program of at

most $O(s)$ lines.

• f can be computed by a NAND circuit of at most $O(s)$ gates.

• f can be computed by a NAND-CIRC straight-line program of at most $O(s)$ line.

(first example) : User-defined procedures 4.1.1

: Theorem 4.1

```
def Proc(a , b) :
        proc_code
        return c
some_code
f = Proc(d , e)
some_more_code
```

بیان می دارد که اگر مفاهیم def و return و کال کردن procedure (خط هایلایت در شبه کد - قالب یا سینتکس پایه زبان های برنامه نویسی برای تعریف تابع توسط برنامه نویس) را با زبان برنامه نویسی NAND-CIRC خود ترکیب کنیم حاصل زبان برنامه نویسی (NAND-CIRC-PROC) است. به ازای هر برنامه نوشته شده با این زبان برنامه نویسی برنامه ای معادل آن با زبان برنامه نویسی NAND-CIRC وجود دارد که همان چیز هارا محاسبه کند. (البته به این شرط که برنامه درون PROC (proc_code که به زبان NAND-CIRC است) دوباره PROC را call نکند یعنی بازگشتی نباشد چون Theorem 4.1 نقض می شود.)

(second example):Example 4.3

Computing Majority from NAND using syntactic sugar

با استفاده از Theorem 4.1، procedure هارا بهمراه NAND-CIRC پروگرم ها استفاده کردیم که نتیجه بسیار تمیز تر و خوانا تری را دارد.

چون ما می توانیم AND, OR وNOT را با استفاده از NAND ها بنویسیم در نتیجه می توانیم Majority function را هم محاسبه کنیم.

```
def NOT(a):
    return NAND(a , a)
def AND(a , b):
    temp = NAND(a , b)
    return NOT(temp)
def OR(a , b):
    temp1 = NOT(a)
    temp2 = NOT(b)
    return NAND(temp1,temp2)
def MAJ(a , b , c):
    and1 = AND(a , b)
    and2 = AND(a , c)
    and3 = AND(b , c)
    or1 = OR(and1 , and2)
    return OR(or1 , and3)
print(MAJ(0 , 1 , 1))
# 1
```

نوع sugar-free همین مثال:

```
temp = NAND(X[0],X[1])
and1 =  NAND(temp,temp)
temp = NAND(X[0],X[2])
and2 =  NAND(temp,temp)
temp = NAND(X[1],X[2])
and3 =  NAND(temp,temp)
temp1 =  NAND(and1,and1)
temp2 =  NAND(and2,and2)
or1 =  NAND(temp1,temp2)
temp1 =  NAND(or1,or1)
temp2 =  NAND(and3,and3)
Y[0] =  NAND(temp1,temp2)
```

این نشان دهنده همان Theorem 4.1 است که می گوید به ازای هر برنامه NAND-CIRC-PROC یک برنامه NAND-CIRC وجود دارد که همان نتایج را محاسبه کند.

Remark 4.4: وقتی تعداد خط را مقایسه می کنیم منظور شمارش تعداد خطوط نوع sugar-free است که در مثال بالا ۱۲ خط است. (کدی که فقط با زبان برنامه نویسی NAND-CIRC است.)

4.1.3 Conditional statements:

می خواهیم یک شرط با ساختار رو به رو به کمک زبان NAND-CIRC-PROC پیاده کنیم. IF(a, b, c) equals b if $a = 1$ and c if $a = 0$

(در واقع یک ساختار if-then-else است.)

```
def IF(cond , a , b):
    notcond = NAND(cond , cond)
    temp = NAND(b , notcond)
```

```
    temp1 = NAND(a , cond)
    return NAND(temp , temp1)
```

فانکشن if در واقع مانند یک multiplexer عمل می کند. به طوری که در کد بالا Cond سوییچ mux و a و b دو پایه ورودی هستند.
حالا که procedure مربوط به if را ساختیم حالا می توانیم شرط ها را پیاده سازی کنیم.

Theorem 4.6: می گوید که اگر NAND-CIRC- IF را زبانی در نظر بگیریم که if/then/else را در کنار زبان NAND-CIRC آوردیم(مثل کاری که با def و Return) کردیم اگر برنامه ای با این زبان بنویسیم حتما برنامه دیگری با زبان NAND-CIRC (همان "sugar-free") وجود دارد که همان محاسبات را انجام دهد.

(second example): 4.2

ADDITION AND MULTIPLICATION

جمع دو عدد n بیتی،استفاده از نوتیشن کم ارزش ترین بیت:

کد پایتون:

```
def ADD(A , B):
    Result = [0] * (n+1)
    Carry = [0] * (n+1)
```

```
    Carry[0] = zero(A[0])
    for i in range(n):
        Result[ i ] = XOR(Carry[ i ] , XOR(A[ i ] , B[ i ]))
        Carry[ i+1 ] = MAJ(Carry[ i ] , A[ i ] , B[ i ])
    Result[n] = Carry[n]
    return Result
ADD([1 , 1 , 1 , 0 , 0] , [1 , 0 , 0 , 0 , 0]) ;;
# [0 ,0 ,1 ,0 ,0 ,0]
```

Theorem 4.7: در رو به رو شکل رسمی جمع دو عدد n بیتی را مشاهده می کنید

$ADD_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ $\to n \in \mathbb{N}$ (تعداد بیت های دو عدد مجموعا 2n و حاصل با احتساب کری n+1 بیت است)

نتیجتا ثابت می شود که یک عدد ثابت c ≥ 30 وجود دارد که برای هر n ای یک برنامه NAND-CIRC وجود دارد(منظور برنامه sugar-free مانند کد پایین) که حداکثر تعداد خطوطش برای محاسبه ADD_n ، cn است.

سپس از روی ADD_n توسط الگوریتم grade-school ضرب را نیز بدست می آوریم.

Theorem 4.8: در رو به رو شکل رسمی جمع دو عدد n بیتی را مشاهده می کنید

$MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$ for every $n \to$ (تعداد بیت های دو عدد مجموعا 2n و حاصل ضرب نیز 2n بیت است)

نتیجتا ثابت می شود که یک عدد ثابت c وجود دارد که برای هر n ای یک برنامه NAND-CIRC وجود دارد که حداکثر تعداد خطوطش برای محاسبه $MULT_n$ ، cn^2 است.

می توانیم از الگوریتم Karatsuba برای نشان دادن این که یک برنامه NAND-CIRC از ($O(n^{\log_2 3})$ خط برای محاسبه $MULT_n$ وجود دارد استفاده کرد.

در پایین کد sugar-free جمع دو عدد n بیتی را مشاهده می کنید.

```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(Temp[1],Temp[1])
Temp[3] = NAND(X[0],X[2])
Temp[4] = NAND(X[0],Temp[3])
Temp[5] = NAND(X[2],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(Temp[2],Temp[6])
Temp[8] = NAND(Temp[2],Temp[7])
Temp[9] = NAND(Temp[6],Temp[7])
Y[0] = NAND(Temp[8],Temp[9])
Temp[11] = NAND(Temp[2],X[0])
Temp[12] = NAND(Temp[11],Temp[11])
Temp[13] = NAND(X[0],X[2])
Temp[14] = NAND(Temp[13],Temp[13])
Temp[15] = NAND(Temp[12],Temp[12])
Temp[16] = NAND(Temp[14],Temp[14])
Temp[17] = NAND(Temp[15],Temp[16])
Temp[18] = NAND(Temp[2],X[2])
Temp[19] = NAND(Temp[18],Temp[18])
Temp[20] = NAND(Temp[17],Temp[17])
Temp[21] = NAND(Temp[19],Temp[19])
Temp[22] = NAND(Temp[20],Temp[21])
Temp[23] = NAND(X[1],X[3])
Temp[24] = NAND(X[1],Temp[23])
Temp[25] = NAND(X[3],Temp[23])
Temp[26] = NAND(Temp[24],Temp[25])
Temp[27] = NAND(Temp[22],Temp[26])
Temp[28] = NAND(Temp[22],Temp[27])
Temp[29] = NAND(Temp[26],Temp[27])
Y[1] = NAND(Temp[28],Temp[29])
Temp[31] = NAND(Temp[22],X[1])
Temp[32] = NAND(Temp[31],Temp[31])
Temp[33] = NAND(X[1],X[3])
Temp[34] = NAND(Temp[33],Temp[33])
Temp[35] = NAND(Temp[32],Temp[32])
Temp[36] = NAND(Temp[34],Temp[34])
Temp[37] = NAND(Temp[35],Temp[36])
Temp[38] = NAND(Temp[22],X[3])
Temp[39] = NAND(Temp[38],Temp[38])
Temp[40] = NAND(Temp[37],Temp[37])
Temp[41] = NAND(Temp[39],Temp[39])
Y[2] = NAND(Temp[40],Temp[41])
```