รายงาน โครงการ สร้างคอมไพล์เลอร์

โดย

นายตะวัน ภูรัต

รหัสนิสิต 5971414021

เพื่อนำเสนอ

ศาสตราจารย์ ดร.ประภาส จงสถิตย์วัฒนา

รายงานนี้เป็นส่วนหนึ่งของวิชา 2110714 Digital System

ประจำภาคเรียนที่ 1 ปีการศึกษา 2559

Project: Instruction

I define the following language which is easy to compile.

(do s1 s2 ...)	sequential s1 s2 ...
(= a b)	assignment a = b
(== a b)	equality test a == b
(ifelse cond T-action F-action)	if..then..else
(def fun-name formal body)	function definition
(print ex)	print integer to screen

with this simple language I can write a program like this:

```
(def fac n
    (ifelse (== n 0)
        1
        (* n (fac (- n 1)))))
(def main ()
    (print (fac 6)))
```

Write a compiler for this language. Your work is in three parts:

1) specification: Write regular expression for words in this language. Write CFG of this language.
2) parser: Write scanner and parser using Recursive Descent method.
3) code generation: Write a code generator to translate Parse Tree into S-code.

You can extend or change this minimal language to include interesting construct (such as for..loop, or array). You can define your own machine code (you don't have to use S-code).

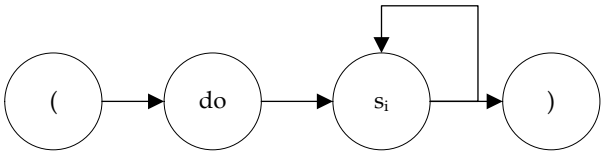
Write your compiler and your report of the work. The report should describe all three parts

with adequate explanation so that I can follow how you design your compiler. You should also give some example of input/output of your system. Do not include the source code of your compiler in the report. Expect the report to be read by an undergraduate student in computer engineering. Email your code to me so I can inspect it. I expect you to spend in average 20-30 man-hours for this project.

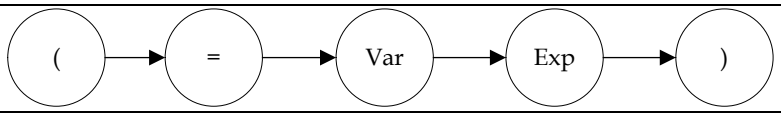
Solution

Part 1)Specification:

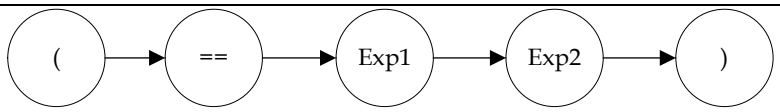
วิเคราะห์จากจากโจทย์ พบว่ามีรูปแบบของคำสั่งที่แตกต่างกัน 6 คำสั่ง ดังนี้

คำสั่งที่ 1	คำสั่ง do (Sequential Computation)
รูปแบบ	(do s1 s2 ..)
ความหมาย	ทำงานตามลำดับเริ่มต้นที่ s1 แล้ว s2 ต่อไปเรื่อยไปจนครบทุกs ที่ระบุ เมื่อ s1,s2 คืองานที่กำหนดให้ทำ
ผังโครงสร้าง	

CFG ย่อย	L=L[L]|λ //rule 0 Language Terminated rule
	L=(do S) //rule 1 Do-instruction rule
	S= s[S] //rule 1.1 sequential tasks repetition rule

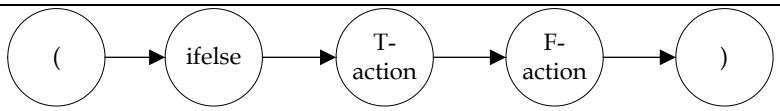
คำสั่งที่ 2	คำสั่งกำหนดค่า (Assignment)
รูปแบบ	(= a b)
ความหมาย	กำหนดให้ a มีค่าเท่ากับ b เมื่อ a เป็นตัวแปรที่ต้องการกำหนดค่า และ b ค่าคงที่หรือเป็นตัวแปรหรือเป็นนิพจน์ใดๆ
ผังโครงสร้าง	

CFG ย่อย	L=L[L]|λ //rule 0 Language Terminated rule
	L=(= A B) // rule 2 Assignment rule
	A=Variable // rule 2.1 Left most operand must be variable
	B=Expression // rule 2.2
	Expression= [Binary_Operator Expression] Integer|Variable // Rule of Expression
	Binary_Operator=+|-|*|/

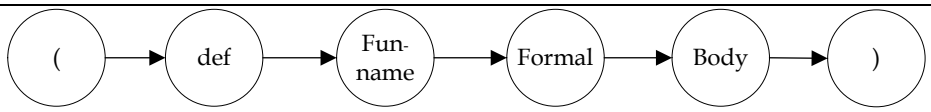
คำสั่งที่ 3	คำสั่งเปรียบเทียบการเท่ากัน (Equality Test)
รูปแบบ	(== a b)
ความหมาย	เป็นการทดสอบว่า a มีค่าเท่ากับ b หรือไม่ เมื่อ a, b เป็นตัวแปร หรือนิจพน์ หรือค่าคงที่
ผังโครงสร้าง	

CFG ย่อย	L=L[L]|λ //rule 0 Language Terminated rule
	L=(== Exp1 Exp2) // rule 3 Equality test rule
	Exp1=Expression
	Exp2=Expression
	Expression = [operator Expression] integer | Variable
	// rule of expression (Prefix style)

คำสั่งที่ 4	คำสั่งทำงานตามเงื่อนไข (If then else)
รูปแบบ	(ifelse cond T-action F-action)

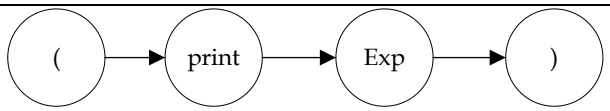
ความหมาย	พิจารณาเงื่อนไข cond ที่กำหนด หากเป็นจริงจะประมวลผล T-action แต่ถ้า cond เป็นเท็จจะประมวลผล F-action แทน
ผังโครงสร้าง	

CFG ย่อย	L=L[L]|λ //rule 0 Language Terminated rule
	L=(ifelse T-action F-action) // rule 1.4 Equality test rule
	T-action=Instruction // rule 1.4.1 Define Action when True
	F-action=Instruction // rule 1.4.2 Define Action when False
	Instrucitons= DC|AC|EC|IC| PC|OP // Define Instruction but DEF is excluded

คำสั่งที่ 5	การนิยามฟังก์ชัน (function definition)
รูปแบบ	(def fun-name formal body)
ความหมาย	เป็นการประกาศสร้างฟังก์ชัน ชื่อ fun-name ขึ้นมาใช้งาน โดยกำหนดให้มีการส่งผ่านข้อมูลตามรูปแบบที่กำหนดไว้ในส่วน formal และให้ฟังก์ชันนี้ทำงานตามที่กำหนดไว้ในส่วนของ body
ผังโครงสร้าง	

CFG ย่อย	L=L[L]|λ //rule 0 Language Terminated rule
	L=(def Fun-name Formal Body) L
	// rule 1.5 define new function rule
	Fun-nam=Name
	Formal=Expression
	Body=Instruction // Nested definition is Excluded

คำสั่งที่ 6	การแสดงค่าจำนวนเต็มออกทางจอภาพ (print integer to screen)

รูปแบบ	(print ex)
ความหมาย	เป็นการแสดงผลของนิพจน์ที่มีค่าเป็นจำนวนเต็มทางหน้าจอภาพ
ผังโครงสร้าง	

CFG ย่อย	L=L[L]|λ //rule 0 Language Terminated rule
	L=(print Exp) // rule 1.6 print expressions rule
	Exp=Expression

จากการวิเคราะห์ทำให้นำมาสรุปสร้างเป็นกฎของภาษาได้ดังนี้

ให้ L คือภาษาที่มีกฎของภาษาดังต่อไปนี้

กฎข้อที่ 0 L=L[L]|λ // First Rule of the Language

กฎย่อยข้อที่ 0.1 Alphabet=a..z|A..Z[Alphabet] //Alphabet definition Rule

กฎย่อยข้อที่ 0.2 Integer=0..9[Integer] //Integer definition Rule

กฎย่อยข้อที่ 0.3 Space = " "[Space] //Spaces definition Rule

กฎย่อยข้อที่ 0.4 Underscore = "_"[Underscore] //Underscores definition Rule

กฎย่อยข้อที่ 0.5 Name=Alphabet|Underscore[|number] // Naming rule

กฎย่อยข้อที่ 0.6 Var=Variable // Var is short name of Variable Rule

Variable=Name // variable must follow naming rule

กฎย่อยข้อที่ 0.7 Exp = Expression // Exp is short name of Expression

Expression=[|BI_Op Exp] Integer|Variable

กฎย่อยข้อที่ 0.8 BI_Op=Binary_Operator //BI_Op is short name of Binary Op.

Binary_Operator=+|-|*|/ // define binary operator

กฎย่อยข้อที่ 0.8 Instruction = DC|AC|EC|IC|PC // Nest Def instruction is Excluded

กฎข้อที่ 1 L=DC|AC|EC|IC|DEF|PC[L] // Program Rule

กฎข้อที่ 1.1 DC = (do S) // Do sequential task rule

กฎย่อยข้อที่ 1.1.1 S= s[S] // more sequential tasks rule

กฎข้อที่ 1.2 AC = (=Var Exp) // Assignment Command Rule

กฎข้อที่ 1.3 EC=(== Exp1 Exp2) // Equality Test Rule

กฎย่อยข้อที่ 1.3.1 Exp1=Expression

กฎย่อยข้อที่ 1.3.2 Exp2=Expression

กฎข้อที่ 1.4 IC = (ifelse Cond T-action F-action) // If else rule

กฎย่อยข้อที่ 1.4.1 Cond=EC //Equality test rule

กฎย่อยข้อที่ 1.4.1 T-action=Instruction

กฎย่อยข้อที่ 1.4.2 F-action=Instruction

กฎข้อที่ 1.5 DEF=(def Fun-name formal body) // Function definition Rule

กฎย่อยข้อที่ 1.5.1 Fun-name = Name // fun name must follow naming

กฎย่อยข้อที่ 1.5.2 Formal = ([Expression[Expression]]) // parameter must be expression

กฎย่อยข้อที่ 1.5.3 Body=Instruciton[Body] // Nested definition is excluded

กฎข้อที่ 1.6 PC=(print Exp) // Display an expression Rule

Part 2) Lexical Analysis (Scanner) & Parser:

2.1 Lexical Analysis (Scanner)

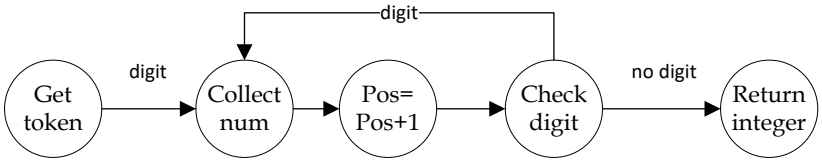
ในส่วน Lexical Analysis นี้เป็นการสร้าง scanner เพื่อแยกตัวอักษรจากแฟ้มเป้าหมาย (source file) มาประกอบเป็น Token ของภาษา โดยดึงตัวอักษรขึ้นมาพิจารณาทีละตัวอักษรซึ่งการทำงานเป็นดังนี้

เริ่มต้น scan ที่ลำดับแรก(ซ้ายสุดในบรรทัด) หาก อักขระที่ scanได้ ไม่เป็น Null ให้พิจารณาตามกรณีดังต่อไปนี้

2.1.1 กรณี เป็นตัวเลข กล่าวคือตัวอักษรที่ได้เป็นสมาชิกในเซต {‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’}

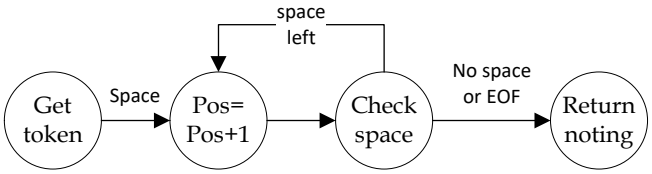
ให้ทำการเช็คล่วงหน้า (advance) หากเป็นตัวเลข ให้ทำการสะสมตัวเลข และทำ advanceตัวอักษรต่อไป

หากไม่ใช่ ให้ถือว่าจบชุดตัวอักษระ ให้Return Token เป็น Integer และค่าคือ ตัวเลขที่สะสมได้ พร้อมทั้งปรับปรุง position ถัดไป ของ line ให้สอดคล้อง



รูปที่ 1 การจัดการตัวเลข

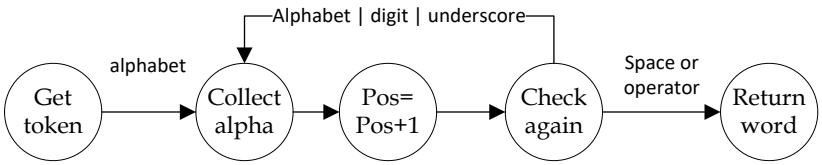
2.1.2 กรณี เป็น space ให้อ่านอักขระถัดไป หากยังคงเป็น space ก็ทำเช่นนี้ แต่หากพบว่าไม่ใช่ space แล้วก็จะไม่ส่งกลับค่า Token กลับแต่อย่างใด มีเพียงการปรับปรุง position ชี้ไปยัง Token ถัดไปให้สอดคล้องเท่านั้น



รูป 2 การจัดการ space

2.1.3 กรณีเป็นเครื่องหมาย = ให้อ่านอักษรถัดไปถ้าเป็น= ให้Return Type เป็น คำสั่ง เปรียบเทียบการเท่ากัน แต่ถ้าไม่ใช่ ให้ส่ง Return Type เป็น คำสั่งกำหนดค่า

2.1.4 กรณีเป็นตัวอักษร ให้อ่านอักขระถัดไป หากไม่ใช่ space เครื่องหมาย ให้ทำการสะสมคำ แต่ถ้าเป็น space หรือเครื่องหมาย ให้ถือว่าได้คำครบถ้วน ให้ส่งคำที่สะสมได้ กลับคืน มี Token Type เป็น keyword นำไปตรวจกับคำสงวน หากตรงกันให้แทนด้วยคำสงวาน แต่ถ้าไม่ตรง ให้ถือเป็นตัวแปร เทียบกับตัวแปรในตาราง แล้วสร้าง Token Type ที่สอดคล้อง



รูปที่ 3 การจัดการกับสายอักษร

ตารางที่ 1 คำสงวนได้และToken

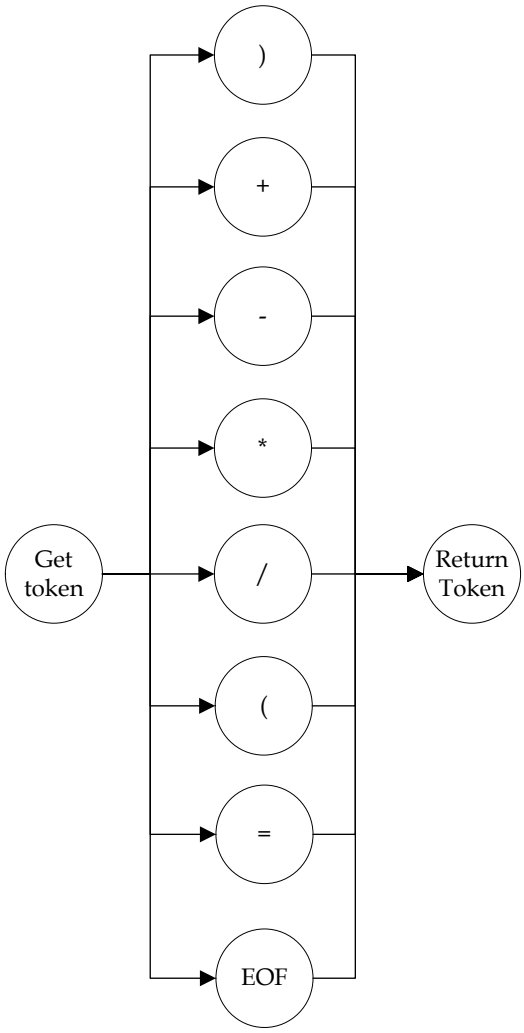
คำสงวน	Token ส่งกลับ	
	Type	Representation
def	DEF	def
do	DO	do
Ifelse	IFELSE	ifelse
print	PRINT	print
=	ASSIGN	=
==	EQT	==

การจัดการ เครื่องหมาย = และ == ทำโดย เมื่ออ่านพบเครื่องหมาย = ให้ทำการ advance ไป 1 ครั้ง และตรวจสอบเครื่องหมาย = อีกครั้ง ถ้าไม่ใช้ ให้ส่ง Token กลับเป็นเครื่องหมาย = หรือการกำหนดค่า (Assignment) แต่ถ้าใช้ให้ส่งเครื่องหมาย == หรือการทดสอบความเท่ากัน (Equality Test)

2.1.5 กรณีอื่นๆ ให้ส่งคืนค่า Token Type และ Token represent เป็นดังตารางต่อไปนี้

ตารางที่ 2 ความสัมพันธ์ของตัวอักขระกับ Token

อักขระที่	Token ส่งกลับ	
scan พบ	Type	Representation
+	PLUS	+
-	MINUS	-
*	MULTIPLY	*
(	LPARM	(
)	RPARM	)



รูปที่ 4. การจัดการกรณีทั่วไป

จากที่กล่าวมานำไปสร้างเป็นโปรแกรมด้วยภาษา python และทำการทดสอบ Lexical Analysis ได้ผลทดลองดังรูปที่

```
Lexer was created from: (def fac n

 Parser phase  Token((, '(')
 Parser phase  Token(def, 'def')
 Parser phase  Token(WORD, 'fac')
 Parser phase  Token(WORD, 'n')

Lexer was created from:     (ifelse (== n 0)

 Parser phase  Token((, '(')
 Parser phase  Token(ifelse, 'ifelse')
 Parser phase  Token((, '(')
 Parser phase  Token(==, '==')
 Parser phase  Token(WORD, 'n')
 Parser phase  Token(INTEGER, 0)
 Parser phase  Token(), ')')

Lexer was created from:        1
>>>
```

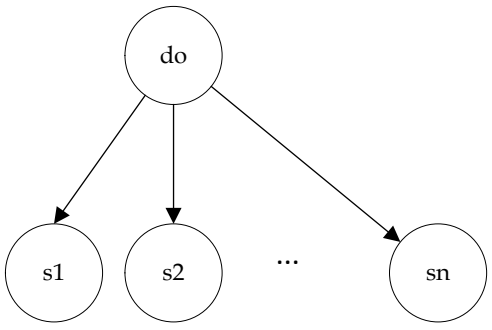
รูปที่ 5 ภาพแสดงผลการทดสอบ Scannerและ parser กับโปรแกรมตัวอย่าง

2.2 การสร้าง Parse Tree

การสร้าง Parse Tree เป็นการสร้างต้นไม้เพื่อนำมาใช้ในการจัดลำดับการทำงานของโปรแกรม โดยลำดับในการคำนวณมีลักษณะเป็น **Deep First Search**

2.2.1 การสร้าง parse tree ของคำสั่ง do

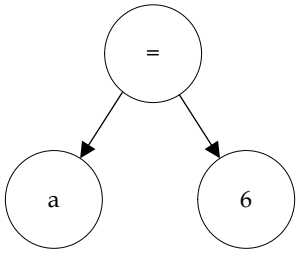
คำสั่ง (do s1 s2 ... sn) เป็นการทำงานตามคำสั่งย่อย s1 จบ แล้วทำ s2 ต่อไป จนไปถึง sn ตามลำดับ มีลักษณะเป็น n-nary tree ดังนั้น หากเขียน node s1 s2 และ sn แทน parse tree ของ คำสั่งย่อยๆ s1,s2, .. sn แล้ว เราจะสร้าง parse tree ของคำสั่ง do ได้ดังนี้



รูปที่ 6 **Parse Tree** ของคำสั่ง **do**

2.2.2 การสร้าง Parse Tree สำหรับคำสั่งกำหนดค่า (Assignment)

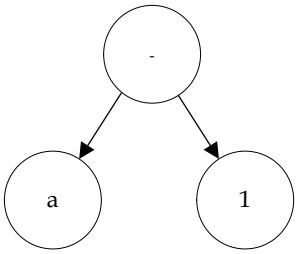
ตัวอย่างเช่นคำสั่ง (= a 6) ในขณะทำงาน scanner เจอเครื่องหมาย = ซึ่งต้องการอีก 2 Term คือต้องเจอ a และเจอ 6 ก็จะสร้าง Tree ได้ดังรูป



รูปที่ 7 Parse Tree สำหรับคำสั่งกำหนดค่า

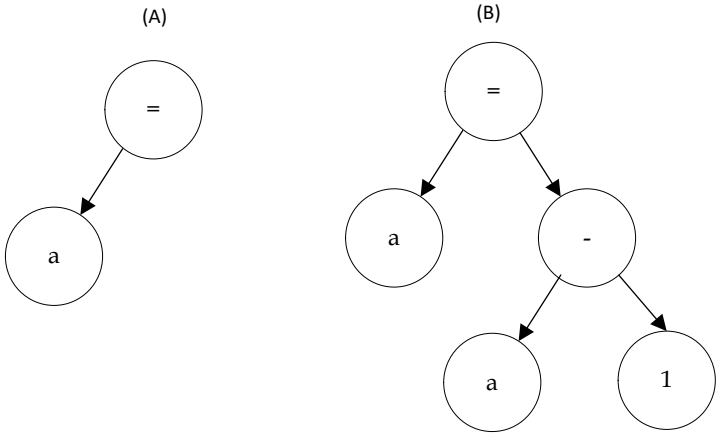
การสร้าง parse tree สำหรับคำนวณนิพจน์

การคำนวณในกรณีภาษานี้ จะมี คำสั่งที่เป็น Binary Operator คือจะต้องมี 2 เทอมเป็นอย่างน้อยจึงจะคำนวณได้ ตัวอย่างเช่น (- a 1) เมื่อพบเครื่องหมาย – ก็ต้องการ อีก 2 เทอม คือตัวตั้งและตัวกระทำ ซึ่งก็คือ a และ 1 เมื่อ scanner ได้สิ่งที่ต้องการครบ ก็จะสร้าง parse Tree ได้ดังรูป



รูปที่ 8 Parse Tree ของคำสั่ง (- a 1)

และเมื่อนำมารวมกับคำสั่ง assign ก็จะทำให้คำสั่งมีความซับซ้อนขึ้นไปอีก การสร้าง pasrse Tree ก็จะเป็นไปตามลำดับดังนี้

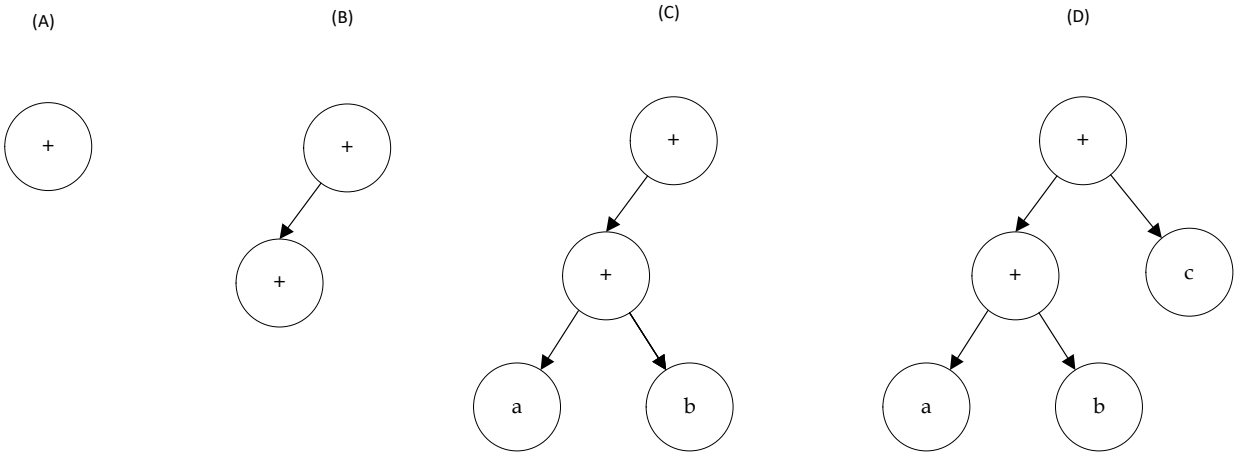


รูปที่ 9 การคำนวณที่มีมากกว่า 2 เทอม เช่น (= a (- a 1)) หรือ a=a-1

Scanner จะเริ่มสร้าง Token ‘(‘, ‘+’, ‘(‘, ‘+’, ‘a’, ‘b’, ‘)’, ‘c’, และ ‘)’ ตามลำดับจากซ้ายไปขวา

แต่การสร้าง Parse Tree จะต้องสร้างให้สอดคล้องกับลำดับการคำนวณ

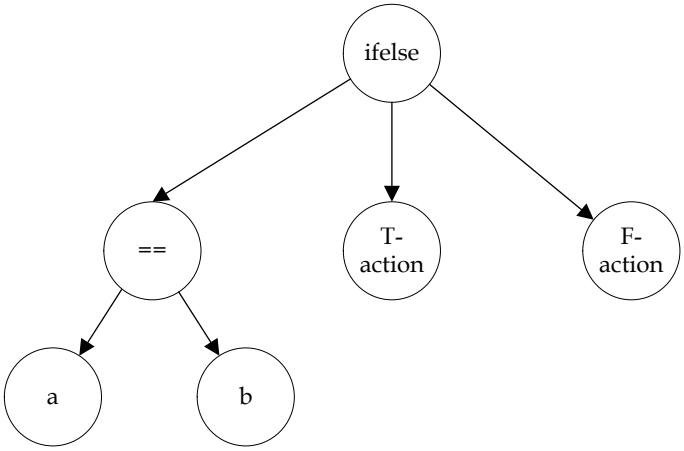
จะได้ลำดับดังนี้



รูปที่ 10 ลำดับขั้นในการสร้าง Parse Tree สำหรับ (+ (+ a b) c)

2.2.3 การสร้าง Parse Tree สำหรับ คำสั่ง เปรียบเทียบ

เนื่องจากคำสั่งเปรียบเทียบ มีการทำงานที่ไม่เป็น Binary กล่าวคือ มีการทำงานเป็น Ternary การสร้างparse Tree จึงมีลักณะเป็น 3 ทางดังรูป

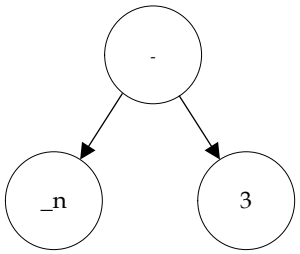


รูปที่ 11 Parse Tree สำหรับคำสั่ง IF ELSE

2.2.4 การสร้าง parse Tree สำหรับคำสั่ง def (Function definition)

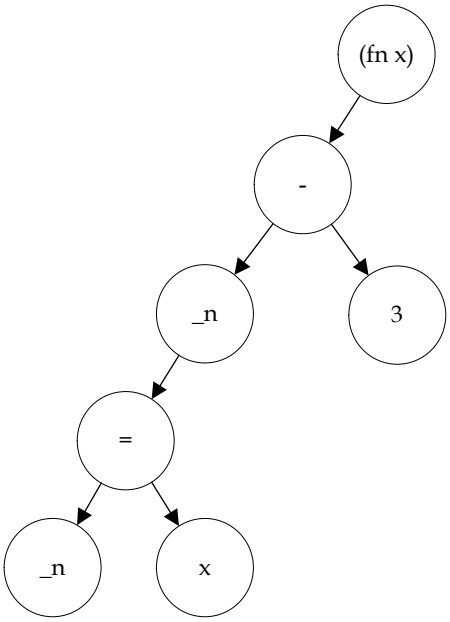
การสร้าง def เป็นการกำหนด abstract instruction เป็นสร้างคำสั่งหรือฟังก์ชั่นใหม่โดยการนำหลายๆคำสั่งมาประกอบต่อเนื่องการ เมื่อเรียกใช้คำสั่งใหม่นี้ ก็จะมีการทำงานตามการทำงานที่ได้กำหนดไว้ และเมื่อทำงานเสร็จ ก็จะส่งค่ากลับมายังจุดที่เรียกใช้ การเรียกใช้จึงเหมือนการต่อกิ่งของต้นไม้ ได้ต้นไม้ที่สร้างไว้นั่นเอง

(def fn n (n-3)) เป็นการสร้างฟังก์ชัน fn(n)=n-3 จะเห็นว่าส่วนคำนวณค่าคือ n-3 ซึ่งเขียนได้เป็น parse tree ดังรูป



รูปที่ 12 Parse Tree ย่อยเฉพาะส่วนภายในของ function

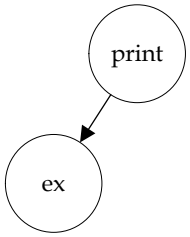
แต่การทำงานจริงๆนั้น เมื่อเรียกใช้ฟังก์ชั่นนี้จำเป็นต้องมีการส่งค่าเข้าไปยังฟังก์ชั่น เช่น ส่งค่า x ไปให้ เสมือนว่ากำหนดค่าให้กับตัวแปร _n (ต้องมีการ rename ชื่อตัวแปรให้เป็นๆ local variable เพื่อป้องกันไม่ให้กระทบตัวแปร n ในจุดที่เรียกมาฟังก์ชั่น กำหนดค่าให้กับตัวแปร _n ทำให้สร้างต้นไม้ได้เป็น ดังรูป



รูปที่ 13 Parse Tree เมื่อมีการเรียกใช้ function

2.5 การสร้าง Parse Tree สำหรับคำสั่ง print

คำสั่ง (print ex) เป็นการนำค่าของ expression ไปแสดงผล หากสร้าง parse tree ของ expression ex ได้ก็จะสร้าง Parse Tree ของคำสั่งนี้ได้ดังรูป (เมื่อ node ex แทน Tree ของ expression ex)



รูปที่ 14 Parse Tree ของคำสั่ง print ex

การประกอบ Tree นั้นเพื่อให้เป็นระเบียบแบบแผน จึงกำหนดให้เติมโหนดใหม่ทางด้านซ้ายมือเป็นลำดับแรก

Part 3) Code generator:

Under construction