รายงานการสร้างคอมไพล์เลอร์

โดย

นายธีรภัทร์ วงศ์สุธีรา

5971420821

เสนอ

ศาสตราจารย์ ดร.ประภาส จงสถิตย์วัฒนา

รายงานนี้เป็นส่วนหนึ่งของวิชา 2110714 Digital System

ประจำภาคเรียนที่ 1 ปีการศึกษา 2559

Specification

1.1) Regular expression

แนวคิด เมื่อพิจารณาโจทย์ที่ได้รับจะพบว่าสัญลักษณ์หรือ word ที่เป็น token เฉพาะดังนี้

1) Reserved word ดังนี้ **do**, **ifelse**, **def**, **print**

2) Comparator ดังนี้ **==**, **!=**, **>**, **<**, **>=**, **<=**

3) Math operation, Assignment ดังนี้ **=**, **+**, **-**, *****, **/**

4) Open parentheses คือ **(**

5) Close parentheses คือ **)**

ดังนั้น token อื่นๆ นอกเหนือจาก token เฉพาะข้างต้นคือ

6) Identifier คือ ชื่อตัวแปรหรือชื่อ function เช่น n, abc, a0, fac

7) Number คือ ตัวเลข เช่น 123, 0, 59

จากแนวคิดข้างต้นจึงได้กำหนด regular expression ออกมาเป็นดังนี้

letter = [**a** – **z** **A** - **Z**]

digit = [**0** - **9**]

identifier = letter (letter | digit) *

number = digit $^{+}$

open parentheses = (**(**)

close parentheses = (**)**)

reserved word ifelse = (**ifelse**)

reserved word do = (**do**)

reserved word def = (**def**)

reserved word print = (**print**)

operation = (**=** | **+** | **-** | ***** | **/**)

comparator = (**==** | **!=** | **>** | **<** | **>=** | **<=**)

Tokens และตัวอย่าง lexeme ในภาษานี้

Tokens	Lexemes
identifier	n, temp, var, x1
number	1, 555, 489
open parentheses	(
close parentheses	)
reserved word ifelse	Ifelse
reserved word do	do
reserved word def	def
reserved word print	print
operation	=, +, -, *, /
comparator	==, !=, >, <, >=, <=

1.2) Context-free grammar

แนวคิด เมื่อพิจารณาข้อกำหนดของโจทย์จึงได้ขอสรุปดังนี้

1) Code ของภาษาจะถูกแบ่งเป็น function โดยจะมีกี่ function ก็ได้ ดังนั้นจึงสามารถเขียน grammar ได้ดังนี้

S -> DEF S | lambda

โดยที่ S เป็น start symbol และ DEF เป็น nonterminal ที่สื่อถึง grammar ในการสร้าง function

2) การสร้าง function นั้นจะเริ่มด้วย terminal ดังนี้ **(**, **def**, identifier หลังจากนี้จะตามด้วย parameter ของ function โดยจะแทนด้วย nonterminal ชื่อ FORMAL และส่วนของชุดคำสั่งในการทำงานอื่นๆ โดยจะแทนด้วย nonterminal ชื่อ SEQ และจบด้วย terminal **)** ดังนั้นจะสามารถเขียนเป็น grammar ได้ดังนี้

DEF -> (def iden FORMAL SEQ)

3) ส่วน parameter ของ function หรือ FORMAL นั้นสามารถเป็นได้ 2 กรณีคือ มีหรือไม่มี parameter ซึ่งกรณีที่ไม่มี parameter จะถูกเขียนด้วย terminal **()** และในส่วนที่มี parameter จะแทนด้วย terminal iden ดังนั้นจึงสามารถเขียน grammar ออกมาได้ดังนี้

FORMAL -> () | iden

4) ส่วนของคำสั่งถูกแทนด้วย SEQ โดยคำสั่งต่างๆ ที่สามารถเป็นไปได้ จะเขียนอยู่ในรูป nonterminal หรือ terminal ได้ดังนี้ DO แทนการสร้างคำสั่ง do, ASM แทนการแทนค่าตัวแปรหรือการทำ math operation ต่างๆ, EQ แทนคำสั่ง compare ต่างๆ, IF แทนคำสั่ง ifelse, PR แทนคำสั่ง print และ number แทนตัวเลขต่างๆ ซึ่งแต่ละ nonterminal จะต้องเริ่มต้นด้วย terminal **(** ดังนั้นจึงสามารถเขียน grammar ได้ดังนี้

SEQ -> (DO | (ASM | (EQ | (IF | (PR | number

5) การสร้างคำสั่ง do นั้นจะเริ่มที่ terminal **do** และตามด้วยชุดคำสั่งต่างๆ ได้โดยที่จะต้องมีอย่างน้อย 1 คำสั่ง ดังนั้นจึงแยกออกเป็น nonterminal D เพื่อรับประกันว่าจะพบอย่างน้อย 1 คำสั่ง และการจบชุดคำสั่ง do จะต้องจบที่ terminal **)** ดังนั้นสามารถเขียนเป็น grammar ได้ดังนี้

DO -> do SEQ D

D -> SEQ D |)

6) การสร้างคำสั่งการแทนค่าตัวแปรหรือการทำ math operation จะเริ่มด้วย terminal op ซึ่งก็คือ token operation จาก regular expression และตามด้วย คำสั่งใดๆ ที่คืนค่าเป็นตัวเลขซึ่งถูกแทนด้วย nonterminal ชื่อ INT และสุดท้ายจะจบด้วย terminal **)** ดังนั้นจึงสามารถเขียนเป็น grammar ได้ดังนี้

ASM -> op INT INT)

7) การสร้างคำสั่งเปรียบเทียบหรือ compare ต่างๆ จะเริ่มที่ terminal cmp ซึ่งก็คือ token comparator จาก regular expression และตามด้วย คำสั่งใดๆ ที่คืนค่าเป็นตัวเลขซึ่งถูกแทนด้วย nonterminal ชื่อ INT และสุดท้ายจะจบด้วย terminal **)** ดังนั้นจึงสามารถเขียนเป็น grammar ได้ดังนี้

EQ -> cmp INT INT)

8) การสร้างคำสั่ง ifelse จะเริ่มที่ terminal **ifelse** และตามด้วยเงื่อนไข โดยจะถูกแทนด้วย nonterminal EQ และตามด้วยชุดคำสั่งต่างๆ ใน then clause และ else clause ซึ่งจะถูกแทนด้วย nonterminal SEQ ทั้งคู่ และปิดท้ายด้วย terminal **)** ดังนั้นจึงเขียนเป็น grammar ได้ดังนี้

IF -> ifelse EQ SEQ SEQ)

9) การสร้างคำสั่ง print จะเริ่มที่ terminal **print** และตามด้วยคำสั่งใดๆ ที่คืนค่าเป็นตัวเลขและปิดท้ายด้วย terminal **)** ดังนั้นจึงสามารถเขียนเป็น grammar ได้ดังนี้

PR -> print INT)

10) การเรียกใช้งาน function จะเริ่มต้นด้วยชื่อ function ที่ต้องการจะเรียกใช้งาน ซึ่งจะถูกเขียนแทนด้วย terminal iden และตามด้วยคำสั่งใดๆ ที่คืนค่าเป็นตัวเลขและปิดท้ายด้วย terminal **)** ดังนั้นจึงสามารถเขียนเป็น grammar ได้ดังนี้

FUNC -> iden INT)

11) คำสั่งที่คืนค่าเป็นตัวเลขได้แก่ การแทนค่า, การเรียกใช้งาน function ซึ่งก็คือ nonterminal ASM และ FUNC ซึ่งจะต้องเริ่มต้นด้วย terminal **(** นอกจากนี้ terminal iden และ number ก็ให้ค่าตัวเลขเช่นเดียวกัน ดังนั้นจึงสามารถเขียน grammar ได้ดังนี้

INT -> (ASM | (FUNC | iden | number

จากข้อสรุปข้างต้นจึงสามารถนำมาเขียนสรุปเป็น context-free grammar ได้ดังนี้

S -> DEF S | lambda

DEF -> (def iden FORMAL SEQ)

FORMAL -> () | iden

SEQ -> (DO | (ASM | (EQ | (IF | (PR | number

DO -> do SEQ D

D -> SEQ D |)

ASM -> op INT INT)

EQ -> cmp INT INT)

IF -> ifelse EQ SEQ SEQ)

PR -> print INT)

FUNC -> iden INT)

INT -> (ASM | (FUNC | iden | number

Parser

2.1) Scanner

หลักการ อัลกอริทึมสำหรับโปรแกรมเพื่อทำ scanner นั้นสามารถจำแนกเป็นขั้นตอนสรุปได้ดังนี้

1) Scanner จะอ่าน input character ทีละตัวไปเรื่อยๆ จนจบครบทุกตัว

2) Character ที่อ่านเข้ามาแต่ละตัวนั้นจะนำไปพิจารณาดูว่ามีโอกาสเป็น lexeme ของ token อื่นๆ ที่ไม่ใช่ identifier หรือ number หรือไม่

2.1) หากมีโอกาสที่จะเป็น lexeme ของ token ดังกล่าวก็จะทำการอ่านตัวถัดไปเรื่อยๆ จนครบตรงกับคำที่เป็น lexeme ตรงกับ token นั้นๆ ก็จะแปลง lexeme นั้นให้เป็นรหัสของ token เพื่อเตรียมสำหรับตรวจสอบ grammar

2.2) หากไม่มีโอกาสหรือไม่เป็น lexeme ของ token เหล่านั้นก็จะตรวจสอบว่าสามารถเป็น lexeme ของ identifier หรือ number ได้หรือไม่ หากสามารถเป็นไรก็จะแปลง lexeme เหล่านั้นเป็นรหัสของ token แต่ถ้าหากไม่สามารถเป็นได้ Scanner จะฟ้อง Syntax error และหยุดโปรแกรมทันที

โดยโปรแกรมสำหรับทำหน้าที่เป็น Scanner สำหรับ Compiler ที่ถูกเขียนขึ้นจากหลักการข้างต้นนี้อยู่ในไฟล์ชื่อ **ScannerCompiler.java**

ตัวอย่าง Input source code ของภาษานี้

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

เมื่อผ่าน Scanner ที่ถูกพัฒนาจากหลักการข้างต้นแล้ว จะได้ Output token ของตัวอย่างนี้ออกมาเป็นดังนี้

```
(def iden iden
     (ifelse (cmp iden number)
          number
          (op iden (iden (op iden number)))))
(def iden ()
     (print (iden number)))
```

2.2) Parser

หลักการ อัลกอริทึมสำหรับ Parser นั้นถูกแบ่งเป็น 2 จุดประสงค์คือ การตรวจสอบ grammar และการสร้าง parse tree แต่ทั้งสองจุดประสงค์นี้จะอ้างอิงพื้นฐานมากจาก Recursive descent method

1) การตรวจสอบ grammar มีวิธีทำและข้อสังเกตดังนี้

1.1) สำหรับแต่ละ nonterminal จะมี function เป็นของตัวเองเพื่อให้ถูกเรียกใช้งาน โดยภายใน function จะอ้างอิงตาม context-free grammar

1.1.1) ถ้าหาก production rule ผลิต nonterminal แล้วภายใน function ก็จะเรียก function ของ nonterminal นั้นๆ ตามที่ production rule สามารถผลิตได้

1.1.2) ถ้าหาก production rule ผลิต terminal แล้วภายใน function จะทำการ match หรือจับคู่ token ที่กำลังอ่านอยู่ ณ ขณะนั้นกับ token จาก context-free grammar ว่าตรงกันหรือไม่ ถ้าหากว่าตรงก็จะทำการอ่านตัวถัดไป แต่ถ้าหากว่าไม่ตรงก็จะทำการฟ้องข้อผิดพลาดและหยุดโปรแกรมทันที

ยกตัวอย่างเช่น ถ้าหาก Context-free grammar เป็นดังนี้

EQ -> cmp INT INT) // โดยที่ EQ, INT INT เป็น nonterminal และ cmp,) เป็น terminal

จะสามารถเขียนโปรแกรมสำหรับตรวจสอบ grammar ได้ดังนี้

```
EQ() {
     match('cmp');
     INT();
     INT();
     match(')');
     return;
}
```

1.2) nonterminal ใดๆ ที่มี production เป็น lambda ได้นั้น ใน function ของ non-terminal นั้นๆ จำเป็นที่จะต้อง look-ahead ไป 1 ครั้งเพื่อให้ทราบว่าควรจะหยุดโปรแกรมหรือไม่

1.3) nonterminal ใดที่มี production ได้มากกว่า 1 รูปแบบ จำเป็นที่จะต้องตรวจสอบ token ณ ตำแหน่งปัจจุบันว่าเป็น token ใด เพื่อที่จะสามารถทราบได้ว่าควรจะผลิต production rule ตัวใดออกมา

1.4) โปรแกรมจะอ่าน token ทีละตัวตั้งแต่จบครบทุก token โดยจะเริ่มการ Recursive ที่ start symbol S หรือ function S() และจะทำตามวิธีข้างต้น

โปรแกรมสำหรับการตรวจสอบ grammar ตามหลักการข้างต้นนี้ ถูกเขียนอยู่ในไฟล์ชื่อ **GrammarCompiler.java**

2) การสร้าง Parse Tree มีวิธีทำและข้อสังเกตดังนี้

2.1) ในการสร้าง parse tree นั้นจะใช้ Stack ทั้งหมด 2 Stack ในการแยกเก็บ operation และ operand

2.2) การสร้าง parse tree จะสามารถทำไปพร้อมๆ กับการตรวจสอบ grammar ได้ ดังนั้นโปรแกรมจะทำงานคล้ายๆ กัน คือใช้ประโยชน์จากการเขียนโปรแกรมแบบ recursive และอ่าน input token ทีละตัวจนหมด

2.3) เมื่อพบ token ที่เป็นประเภท operation จะทำการ push lexeme ของ token นั้นๆ ลงไปใน stack สำหรับเก็บ operation ซึ่ง token ที่เป็นประเภท operation สำหรับภาษานี้ได้แก่ do, operation, comparator, ifelse และ identifier ที่เกิดจาก nonterminal FUNC เนื่องจากเป็นการเรียกใช้ function

2.4) เมื่อพบ token ที่เป็นประเภท operand จะทำการ push lexeme ของ token นั้นๆ ลงไปใน stack สำหรับเก็บ operand ซึ่ง token ที่เป็นประเภท operand สำหรับภาษานี้ได้แก่ number, identifier และ **()** ที่เกิดจาก nonterminal FORMAL

2.5) เมื่อพบ token **)** จะทำ pop stack ที่เก็บ operation เพื่อทำการสร้าง tree โดยขึ้นกับแต่ละ operation ที่ได้จากการ pop ดังนี้

2.5.1) ถ้าหาก operation คือ token operation จะทำการ pop stack ที่เก็บ operand ทั้งหมด 2 ครั้งและทำการต่อเป็นโครงสร้างต้นไม้ดังนี้ (operation operand operand) และ push โครงสร้างต้นไม้นี้กลับลงไปใน stack ที่เก็บ operand

2.5.2) ถ้าหาก operation คือ token ifelse จะทำการ pop stack ที่เก็บ operand ทั้งหมด 3 ครั้งโดยครั้งแรกคือ else-clause ครั้งที่สองคือ then-clause และครั้งที่สามคือ condition หลังจากนั้นจะทำการต่อเป็นโครงสร้างต้นไม้ดังนี้ (operation condition then-clause else-clause) และ push โครงสร้างต้นไม้นี้กลับลงไปใน stack ที่เก็บ operand

2.5.3) ถ้าหาก operation คือ token print จะทำการ pop stack ที่เก็บ operand ทั้งหมด 1 ครั้งและทำการต่อเป็นโครงสร้างต้นไม้ดังนี้ (operation operand) และ push โครงสร้างต้นไม้นี้กลับลงไปใน stack ที่เก็บ operand

2.5.4) ถ้าหาก operation คือ token identifier ที่เป็นชื่อ function จะทำการ pop stack ที่เก็บ operand ทั้งหมด 1 ครั้งและทำการต่อเป็นโครงสร้างต้นไม้ดังนี้ (call operation operand) และ push โครงสร้างต้นไม้นี้กลับลงไปใน stack ที่เก็บ operand

2.5.5) ถ้าหาก operation คือ token def จะทำการ pop stack ที่เก็บ operand ทั้งหมด 3 ครั้งโดยครั้งแรกคือ body ของ function ครั้งที่สองคือ parameter ของ function และครั้งที่สามคือชื่อของ function หลังจากนั้นจะทำการต่อเป็นโครงสร้างต้นไม้ดังนี้ (fun name body) ซึ่ง tree นี้คือ parse tree ของ function หนึ่งใน source code

โปรแกรมสำหรับการสร้าง Parse Tree ถูกเขียนอยู่ในไฟล์ชื่อ **ParseTreeCompiler.java**

ตัวอย่าง Input token จาก Scanner

```
(def iden iden
     (ifelse (cmp iden number)
          number
          (op iden (iden (op iden number)))))
(def iden ()
     (print (iden number)))
```

เมื่อผ่าน Parser ที่ถูกพัฒนาจากหลักการข้างต้นแล้ว จะได้ Output parse tree ของตัวอย่างนี้ออกมาเป็นดังนี้

```
(fun fac (ifelse (== #1 0)(return 1)(return (* #1 (call fac
(- #1 1 ))))))
(fun main (print (call fac 6)))
```

Code Generation

การแปลงจาก Parse Tree เป็น S-code มีหลักการและข้อสังเกตดังนี้

1) โปรแกรมจะอ่าน input เข้ามาทีละ node ของ parse tree และจะใช้ประโยชน์จากการ recursive เพื่อลดความซับซ้อนในการแปลง parse tree เป็น S-code

2) การเริ่มต้นของโปรแกรมนั้นเป็นที่แน่นอนว่าจะพบกับ input ในลักษณะ **(fun function_name** ... โดยที่ **function_name** คือชื่อของ function ของ tree นั้นๆ ดังนั้นก็จะสามารถส่ง output S-code ออกมาก่อนคือ **:function_name**

3) หลังจากผ่านส่วนแรกมาแล้ว ก็จะเริ่มการ recursive โดย function ที่ใช้ในการ recursive นั้นจะมีหน้าที่จัดการกับ tree ที่ได้รับมาทีละ 1 level โดยจะจำแนกคำสั่งต่างๆ ที่พบ เพื่อที่จะสามารถจัดการแปลง parse tree ได้ถูกต้องตามโครงสร้างของคำสั่งนั้นๆ ซึ่งสามารถจำแนกคำสั่งต่างๆ ได้เป็น

3.1) **ifelse** จะต้องทำการเรียก function recursive ทั้งหมด 3 ครั้ง คือ ส่วนของ condition, then-clause และ else clause เนื่องจากแต่ละส่วนสามารถเป็น tree ที่ซับซ้อนด้านใน ดังนั้นจึงเรียก function recursive เพื่อจัดการ tree ด้านในให้เสร็จก่อน ซึ่งคำสั่ง ifelse จำเป็นที่จะต้องจดจำบรรทัดในการ jump คำสั่งสำหรับแต่ละ condition ด้วย

3.2) **return** จะทำการ recursive เพียงครั้งเดียว เนื่องจากการค่าที่ return อาจจะเป็น tree ได้ และจะให้ S-code เป็น **ret.tree** เมื่อ tree คือลำดับของ tree

3.3) **call** จะทำการ recursive 1 ครั้ง เพราะค่า parameter ที่ส่งไปในการเรียก function อาจจะเป็น tree ได้ โดยจะต้องทำการจดจำชื่อ function ที่เรียกด้วย เนื่องจากเมื่อแปลงเป็น S-code แล้วจะต้องมี output เป็น **call.function_name** โดยที่ **function_name** คือชื่อ function ที่ถูกเรียก

3.4) **print** จะทำการเรียก recursive 1 ครั้ง เพราะค่าที่ print อาจเป็น tree ได้ และจะให้ output S-code เป็น **sys.1**

3.5) **math operation** หรือ **comparator** จะทำการเรียก recursive 2 ครั้ง เนื่องจากคำสั่งเหล่านี้จะใช้ operand ทั้งหมด 2 ตัว และเมื่อเรียก recursive แล้วก็จะใช้ output S-code เป็น math operation หรือ comparator ต่างๆ ตามที่พบ เช่น **mul** ในกรณีที่พบ operation *****, **eq** ในกรณีที่พบ comparator **==**

3.6) **number** หรือ **identifier** จะไม่ทำการเรียก recursive ต่อ เนื่องจากเป็น leaf node ของ tree แล้ว ดังนั้นสามารถให้ output S-code ออกมาได้เลย เช่น **lit.10** ในกรณีที่พบ number มีค่า 10, **get.1** ในกรณีที่พบ identifier ตัวแรก

โปรแกรมการแปลง Parse Tree เป็น S-code นั้นถูกเขียนอยู่ในไฟล์ชื่อ **ScodeCompiler.java**

ตัวอย่าง Input Parse Tree

```
(fun fac (ifelse (== #1 0)(return 1)(return (* #1 (call fac
(- #1 1 ))))))
(fun main (print (call fac 6)))
```

ตัวอย่าง Output S-code

```
:main
     fun.1
     lit.6
     call.fac
     sys.1
     ret.1
:fac
     fun.1
     get.1
     lit.0
     eq
     jf.16
     lit.1
     ret.2
     jmp.24
:L16
     get.1
     get.1
     lit.1
     sub
     call.fac
     mul
     ret.2
:L24
     ret.2
```