Verbal programming language interface

ชัยภัทร จุลศรี
5731025921
adelaide.8000@gmail.com

ใน report ฉบับนี้ ผมจะพูดถึงภาษาที่ผมสร้างสำหรับเขียนโปรแกรมภาษาพูดขึ้นมา โดยจะแบ่งออกเป็นส่วนหลักๆ สองส่วน คือ เกี่ยวกับ Parser ของภาษา และ เกี่ยวกับ Grammar ของภาษาเอง

สำหรับ parser ผมใช้การ parse แบบ LL(1) โดยเขียนโค้ดภาษา Haskell อ่าน grammar จาก text file ไป parse text file อีกไฟล์ ในการหา first set ผมเขียนตามอัลกอริทึมเลย แต่ follow set ต้องมีการแปลงที่ซับซ้อนหน่อย แต่ก็ไม่ยากเกินไปนัก parsing table ก็เขียนตามไปเลย ส่วนนี้ไม่มีอะไรยาก นอกจากการ parse Id กับ Filename ที่ผมกำหนดให้เป็น wildcard ทำให้ต้อง hardcode การ parse wildcard ลงไปในโค้ดด้วย

ผมได้อัพโหลด source code ไว้ที่ https://github.com/Fulmene/ll1-parser

Grammar ที่ผมจะเขียนต่อไปนี้ จะเป็น Grammar ของภาษาพูดที่จะถูก generate เป็นภาษาแบบ C-like ที่ไม่มี type ของตัวแปร ในภาษามี feature ดังต่อไปนี้

- มีการ include โค้ดจากไฟล์อื่นมาใช้
- มีการเขียน subroutine เพื่อทำงานย่อย โดยในภาษานี้จะเรียกรวมๆ ทั้งหมดว่า function
- มีการแบ่งโค้ดเป็น block โดยภายใน block จะประกอบด้วย statement หลายๆ อันรวมกัน
- statement แบ่งออกเป็นประเภทย่อยๆ ดังนี้
  - expression ซึ่งประกอบด้วย
    - boolean expression
      - มี boolean operator พื้นฐาน คือ and or not
      - มี operator สำหรับเปรียบเทียบตัวเลข ได้แก่ เท่ากับ ไม่เท่ากับ น้อยกว่า น้อยกว่าหรือเท่ากับ มากกว่า มากกว่าหรือเท่ากับ
    - arithmetic expression
      - มี operator สำหรับการบวก ลบ คูณ หาร หารเอาเศษ
      - ตัวเลขสามารถมีจุดทศนิยม และค่า exponent ได้ หรือไม่มีก็ได้
    - function call
    - identifier โดดๆ ใช้สำหรับเวลาที่ใช้ expression เป็น argument ของ function call หรือเป็น return value
  - assignment สำหรับ assign ค่าจาก expression ไว้ที่ identifier
  - return statement สำหรับส่งค่ากลับไปยัง caller ของ function
  - control flow statement
    - if-elsif-else สำหรับ conditonal branching
    - while และ for สำหรับ loop

Grammar ที่จะเห็นต่อไปนี้เป็นอันเดียวกับที่ผมให้ parser ของผมอ่าน แล้วให้มันคำนวณ first set, follow set และ parsing table ให้ แล้ว parse ด้วยวิธี LL(1) แน่นอนว่าการทำแบบนี้ performance จะไม่ดีเพราะต้องอ่าน grammar ใหม่ทุกรอบ ถ้าผมว่างอาจจะกลับมาแก้ให้โปรแกรม “คอมไพล์” grammar เป็น format ที่อ่านง่ายกว่านี้ก่อน จะได้อ่าน grammar ทีเดียว แล้วรอบต่อไปก็อ่านจากไฟล์ที่คอมไพล์มา

สำหรับตัว grammar เองนั้น ผมพยายามเขียนให้คนอ่านเข้าใจง่ายที่สุด ส่วนใหญ่แล้วจะไม่มีปัญหา นอกจากส่วนที่เป็น infix binary operation ทั้งหลาย ที่ถ้าเขียนตรงๆ จะเกิด left recursion แล้วใช้ LL(1) parsing ไม่ได้ ทำให้ส่วนที่อ่านยากมีอยู่แค่นั้น

Grammar ของภาษานี้มีเนื้อหาดังต่อไปนี้

<Prog> ::= <Inc> <Prog> | <Func> <Prog> | empty

<Inc> ::= include <File> stop
<File> ::= <Filename> <Files>
<Files> ::= <Filename> <Files> | empty
<Filename> ::= wildcard

<Func> ::= function <Id> <Param> <Block>
<Param> ::= parameter <Id> <Params> stop | empty
<Params> ::= <Id> <Params> | empty

<Block> ::= block <Stmts> stop
<Stmts> ::= <Stmt> <Stmts> | empty
<Stmt> ::= <Expr> | <Asgn> | <Ret> | <Ctrl>

<Expr> ::= boolean <Bool> | arithmetic <Artm> | <Id> | <Call>

<Bool> ::= <BoolExpr> <Bools>
<Bools> ::= <BoolOp> <BoolExpr> <Bools> | empty
<BoolExpr> ::= <BoolTerm> | <BoolBracket> | <BoolNot>
<BoolBracket> ::= bracket <Bool> stop
<BoolNot> ::= not <Bool> stop
<BoolTerm> ::= true | false | <Comp> | <Id> | <Call>
<BoolOp> ::= and | or

<Comp> ::= compare <Artm> <CompOp> <Artm>
<CompOp> ::= equal | inequal | less | lequal | more | mequal

<Artm> ::= <ArtmExpr> <Artms>
<Artms> ::= <ArtmOp> <ArtmExpr> <Artms> | empty
<ArtmExpr> ::= <ArtmTerm> | <ArtmBracket> | <ArtmNegate>
<ArtmBracket> ::= bracket <Artm> stop
<ArtmNegate> ::= negate <Artm> stop
<ArtmTerm> ::= <Number> | <Id> | <Call>
<ArtmOp> ::= plus | minus | mul | div | mod

<Number> ::= <Integer> <Fraction> <Exponent>
<Integer> ::= <Sign> <NumInit> <Digits> | zero
<Sign> ::= positive | negative | empty
<Fraction> ::= point <Digits> | empty
<Exponent> ::= expo <Integer> | empty
<Digits> ::= <Digit> <Digits> | empty
<NumInit> ::= one | two | three | four | five | six | seven | eight | nine
<Digit> ::= zero | <NumInit>

<Asgn> ::= assign <Id> to <Expr>

<Call> ::= call <Id> <CallParam>
<CallParam> ::= parameter <CallParam1> stop | empty
<CallParam1> ::= <Expr> <CallParams>
<CallParams> ::= <Expr> <CallParams> | empty

<Ret> ::= return <Expr>

<Ctrl> ::= <If> | <Loop>
<If> ::= if <Bool> then <Block> <Elsif> <Else>
<Elsif> ::= elsif <Bool> then <Block> <Elsif> | empty
<Else> ::= else <Block> | empty
<Loop> ::= <While> | <For>
<While> ::= while <Bool> do <Block>
<For> ::= for <Id> from <Artm> to <Artm> do <Block>

<Id> ::= wildcard

ตัวอย่างโปรแกรม ผมขออนุญาตใช้ตัวอย่างคลาสสิกอย่าง “แสดงเลข 1 ถึง 10” ซึ่งเขียนแบบ C-like ดังนี้

```
#include <stdio>
main() {
  for (i=1;i<=10;i++)
    print(i);
}
```

ซึ่งในภาษาของผม จะเขียนได้ดังนี้

```
include stdio stop
function main block
  for i from one to one zero do block
    call print parameter i stop
  stop
stop
```

อีกตัวอย่างที่ผมจะลองให้ดูคือ ฟังก์ชันหาค่าฟิโบนัชชีลำดับที่ n ดังนี้

```
fibonacci(n) {
  a = 0;
  b = 1;
  for (i=2;i<=n;i++) {
    if (i%2 == 0) { a = a+b; }
    else { b = a+b; }
  }
  if (n%2 == 0) { return a; }
  else { return b; }
}
```

เขียนเป็นภาษาผมจะค่อนข้างยาว แต่ได้ดังนี้

```
function fibonacci parameter n stop block
  assign a to arithmetic zero
  assign b to arithmetic one
  for i from two to n do block
    if compare i mod two equals zero then block assign a to arithmatic a plus b stop
    else block assign b to arithmetic a plus b stop
  stop
  if compare n mod two equals zero then block return a stop
  else block return b stop
stop
```

สำหรับโปรแกรมอื่นๆ download source code ไปคอมไพล์แล้วลองเล่นได้ครับ