Compiler Design Lab (CS 511)

Autumn 2021

Assignment 8

**Consider the following context-free grammar G.**

**The non-terminals are:**

N = { AS, BE, D, DL, E, IOS, IS, LS, P, PE, RE, S, SL, TY, VL }

**The terminals are:**

Σ = { + − / == < > ( ) = ; & | ~ ∗ if then else ic fc id int real do while nop print read str},

**The start symbol is P and the production rules are**

P → DL SL //DL -> Data (Declaration) List, SL -> Statement List

DL → DL D | ε //D-> Data (Declaration)

D → VL : TY //VL -> Variable (Name), TY -> (Data) Type

TY → int | real

VL → VL id | id

SL → SL | S //S-> (Final) Statement

S → AS | IS | LS | IOS | nop //AS-> Assignment, IS-> If Statement, LS-> Loop Statement

AS → id = E //E-> (Final) Expression

IS → if BE then SL else SL ; //BE-> Branch Expression

LS → while BE do SL ;

IOS → print PE | read id //IOS-> Input/Output Statement, PE-> Print Expression

PE → E | str

BE → BE ‘|’ BE | BE & BE | ~ BE | ( BE ) | RE //RE-> Relational Expression

RE → E == E | E < E | E > E

E → E + E | E − E | E E | E / E | − E | ( E ) | id | ic | fc ∗

Most of the terminals have their usual meaning e.g. id is an identifier, ic is an integer constant, fc is a floating-point constant, str is a string of characters, nop for no operation, = is an assignment, == is equality, ‘|’ is logical ‘or’, & is logical ‘and’, ~ is logical ‘not’ etc. Both ‘|’ and ‘&’ are left associative. The precedence relations of logical operators are ‘|′ < ‘&′ < ‘~’.

An identifier starts with an English alphabet followed by a sequence of alphabet or decimal digits. An integer constant is a sequence of decimal digits. a floating-point constant is a sequence of decimal digits with a decimal dot in it (.12, 12., 1.2).

A comment starts with ‘//’ and is up to the end of line (‘\ n’). A character string is within a pair of

quotes (”).

1. Write a flex specification of the scanner. The name of the flex specification file should be <reg\_no>.l The command **$ flex <reg no>.l** generate the C code for the scanner in lex.yy.c. Include the header file <reg no>.tab.h++ in the definition section of your flex specification. This header file, created by bison, contains the token names, type of yylval etc.

2. Write bison specification for the grammar to generate a parser. You need to specify the precedence and associativity of boolean and arithmetic operators as these parts of the production rules makes the grammar ambiguous. The specification file name should be <reg no>.y++ (this will generate C++ parser code).

The command **$ bison -d -v <reg no>.y++** generates three files

**<reg no>.tab.c++,**

**<reg no>.tab.h++, and**

**<reg no>.output.**

The output file contains the description of the LALR(1) automaton.

3. The compiled code of <reg no>.tab.c++ and lex.yy.c gives the basic parser. Let the name of the executable file be myParser. Prepare a Makefile for the whole process.

4. To send the assignment for evaluation prepare a zip file with the name <reg no>.6.zip which includes three (3) files: Makfile, <reg no>.l, <reg no>.y++. Others format like rar, tar, tar.xz etc. will not be evaluated. Do not put them under a subdirectory while preparing the zip file.

Sample Input:

// This program computes factorial

n fact i : int

read n

i = 1

fact = 1

while i < n | i == n do

fact = fact \* i

i = i + 1; // Note the ;

from LS → while BE do SL ;

print fact

Sample Output: Accepted