CompSci 330 Assignment 1 LALR(1) Parsing, etc.

Obtain the tar/gzip file ASSIGN1STUD.tar.gz from the assignment directory. Gunzip/untar this file by typing

```
tar -x -z -f ASSIGN1STUD.tar.gz
```

to create a directory ASSIGN1STUD, containing various template files.

If this fails, it is probably because your version of tar does not support the -z option. In this case gunzip and the file separately, by typing

```
gunzip ASSIGN1STUD.tar.gz
tar -x -f ASSIGN1STUD.tar
```

Question 1

Consider the CUP grammar in Appendix 1.

(a) Use the LALR(1) state information in Appendix 2. Construct the action table and goto table from these states.

(5 marks)

(b) Perform an LALR(1) parse of the input in Appendix 3. Display the information in the same style as chapter 2 of the lecture notes.

(10 marks)

(c) Draw the parse tree corresponding to the LALR(1) parse. The root node should correspond to the rule "$START ::= Program $". I would prefer that nodes correspond to grammar rules, rather than terminal and nonterminal symbols.

(10 marks)

(d) Draw the abstract syntax tree built by the actions in the CUP grammar.

(10 marks)

Total 35 Marks

Question 2

Consider the grammar in the directory COMPILE.

A program is composed of a sequence of statements.

What you have to do is write a "compiler" to "compile" a program for this language into Alpha assembly language. It is actually very easy.

Assume we never run out of temporary registers.

We can generate code for expressions by a recursive method

```
public void evalCode( int freeReg );
```

The int parameter freeReg indicates the first free temporary register. For example, if freeReg is 2, then registers $t2, $t3, $t4, ... can be used to evaluate the expression.

The algorithm is:

- For most binary operators:
  - Generate code to evaluate the left operand into freeReg.
  - Generate code to evaluate the right operand into freeReg + 1.
  - Generate an operate instruction with suitable opCode to perform the operation with source operands freeReg and freeReg + 1, and destination freeReg.

```
public void evalCode( int freeReg ) {
     left.evalCode( freeReg );
     right.evalCode( freeReg + 1 );
     Code.instrn( opCode,
          Code.tempReg( freeReg ), Code.tempReg( freeReg + 1 ) );
     }
```

  For example, if freeReg = 0, and the expression is "a * b", we might generate code

```
     ldiq $t0, a;
     ldq $t0, ($t0);
     ldiq $t1, b;
     ldq $t1, ($t1);
     mulq $t0, $t1;
```

- For most prefix operators:
  - Generate code to evaluate the right operand into freeReg.
  - Generate an operate instruction with suitable opCode to perform the operation with source operand freeReg, and destination freeReg.

```
public void evalCode( int freeReg ) {
     right.evalCode( freeReg );
     Code.instrn( opCode,
          Code.tempReg( freeReg ), Code.tempReg( freeReg ) );
     }
```

  For example, if freeReg = 0, and the expression is "- a", we might generate code

```
     ldiq $t0, a;
     ldq $t0, ($t0);
     negq $t0, $t0;
```

- For simple variables:
  - Generate a load immediate instruction to load the address of the variable into freeReg.
  - Generate a load instruction to load the contents of the memory pointed to by freeReg into freeReg.

```
public void evalCode( int freeReg ) {
     Code.instrn( "ldiq", Code.tempReg( freeReg ), ident );
     Code.instrn( "ldq", Code.tempReg( freeReg ),
          Code.indirect( freeReg ) );
     }
```

For example, if freeReg = 1, and the expression is "b", we might generate code

```
ldiq $t1, b;
ldq $t1, ($t1);
```

- For most constants:
  - Generate a load immediate instruction to load the value of the constant into freeReg.

```
public void evalCode( int freeReg ) {
     Code.instrn( "ldiq", Code.tempReg( freeReg ), Code.literal( value ) );
     }
```

For example, if freeReg = 1, and the expression is "24", we might generate code

```
ldiq $t1, 24;
```

- The code generated should leave the result in the register corresponding to freeReg.

We can generate code for statement sequences and statements, by a method

```
public void genCode();
```

- To generate code for statement sequences:
  - Loop, generating code for each component statement.

```
public void genCode() {
     for ( int i = 0; i < size(); i++ ) {
          StmtNode stmt = elementAt( i );
          stmt.genCode();
          }
     }
```

- To generate code for assignment statements:
  - Generate code to load the address of the variable (the left operand) into $t0.
  - Generate code to evaluate the expression (the right operand) into $t1 (freeReg = 1).
  - Generate a store instruction to store the contents of $t1 into the address pointed to by $t0.

```
public void genCode() {
     Code.enter();
     Code.instrn( "ldiq", "$t0", ident );
     expr.evalCode( 1 );
     Code.instrn( "stq", "$t1", "($t0)" );
     Code.exit();
     }
```

(Code.enter() and Code.exit() generate the braces, to provide a local block.)

For example, for the statement "a = b + 24;", we might generate code

```
{
     ldiq $t0, a;
     ldiq $t1, b;
     ldq $t1, ($t1);
     ldiq $t2, 24;
     addq $t1, $t2;
     stq $t1, ($t0);
}
```

- To generate code for "print" statements:

- Generate code to evaluate the parameters into $a0, $a1, $a2, ...
- Generate an instruction to invoke the IO.printf method.

```
public void genCode() {
     Code.enter();
     exprList.evalCode();
     Code.instrn( "bsr", "IO.printf.enter" );
     Code.exit();
     }
```

For example, for the statement “print(“%d\n”, a * b);”, we might generate code

```
     {
          {
               const {
               string:
                    asciiz "%d\n";
                    align;
               } const
               ldiq $t0, string;
          }
          mov $t0, $a0;
          ldiq $t0, a;
          ldq $t0, ($t0);
          ldiq $t1, b;
          ldq $t1, ($t1);
          mulq $t0, $t1;
          mov $t0, $a1;
          bsr IO.printf.enter;
     }
```

- To generate code for “while” statements:
  - Generate a label definition for “while”.
  - Generate code to evaluate the condition into $t0.
  - Generate code to branch to “end” if the condition is false.
  - Generate a label definition for “do”.
  - Generate code to evaluate the substatement.
  - Generate code to branch back to “while”.
  - Generate a label definition for “end”.

```
public void genCode() {
     Code.enter();
     Code.labelDefn( "while" );
     cond.evalCode( 0 );
     Code.instrn( "blbc", "$t0", "end" );
     Code.labelDefn( "do" );
     stmt.genCode();
     Code.instrn( "br", "while" );
     Code.labelDefn( "end" );
     Code.exit();
     }
```

For example, for the statement “while 0 < a do a = a - 1;”, we might generate code

```
     {
     while:
          ldiq $t0, 0;
          ldiq $t1, a;
          ldq $t1, ($t1);
          cmplt $t0, $t1;
```

```
            blbc $t0, end;
        do:
            {
                ldiq $t0, a;
                ldiq $t1, a;
                ldq $t1, ($t1);
                ldiq $t2, 1;
                subq $t1, $t2;
                stq $t1, ($t0);
            }
            br while;
        end:
        }
```

We also need to generate code to allocate space for each variable. Assume our variables are simple identifiers, and all variables are assigned to before they are used.

Assume we have a compile-time environment "env" that represents the set of identifiers seen so far (or at least those as the destination of an assignment).

We can process each statement and statement sequence by a method

```
public void genDeclCode( Env env );
```

For each assignment statement, this method:

- Checks whether the destination identifier is already in the environment.
- If we have not processed the identifier, outputs a label definition for the identifier, and a "quad' directive to allocate space for the identifier.
- Adds the identifier to the environment.

For example, if we use variables a, d, we might generate code

```
a:
    quad 0;
d:
    quad 0;
```

In addition to this, we need to package the code within some text to import support files, specify the entry point, etc. For example, we might generate code

```
entry main.enter;
import "../../IMPORT/callsys.h";
import "../../IMPORT/proc.h";
import "../../IMPORT/callsys.lib.s";
import "../../IMPORT/string.lib.s";
import "../../IMPORT/number.lib.s";
import "../../IMPORT/io.lib.s";
public block main {
    data {
        ... // Code generated by genDeclCode
    } data
    code {
    public enter:
        ... // Code generated by genCode, exprCode
        clr $a0;
        bsr Sys.exit.enter;
    } code
} block main
```

Sample input and generated code is illustrated in appendix 5.

Modify the code for parser.cup, Ylex.jflex and the Node classes so that the compiler reprints programs suitably indented, and generates appropriate Alpha assembly language for the following constructs.

Add in the grammar and code for the "true", "false" literal values (which correspond to the numeric values of 1 and 0). Ensure that they reprint as "true' and "false".

(5 marks)

Add in grammar and code for the "||", "&&", "!", ">", ">=", "!=" operators. Assume (infix) "||" has the lowest precedence, then (infix) "&&", then (prefix) "!", then the (infix, non-associative) relational operators, then (infix) "+" and (infix and prefix) "-", then (infix) "*" and "/".

(10 marks)

Add in grammar and code for "if-then", "if-then-else", and "for" statements.

The syntax for "if-then" statements is "if Expr then Stmt". The syntax for "if-then-else" statements is "if Expr then Stmt else Stmt". The syntax for "for" statements is "for IDENT = Expr to Expr do Stmt".

(20 marks)

Total 35 Marks

Using the shell scripts

Shell scripts to compile the compiler

Basically you just run createcompiler.bash, and fix the errors until your compiler compiles.

createlexer.bash	Run JFlex. Put the error messages in jflex.error. Create Source/grammar/Ylex.java.
createparser.bash	Run CUP. Put the error messages in cup.error. Create Source/grammar/{parser.java,sym.java,parser.states}.
createclass.bash	Run javac. Put the error messages in javac.error. Create class files in Classes directory.
createjar.bash	Run jar. Create the jar file run.jar.
createcompiler.bash	Run the other create*.bash files, to compile the compiler, and generate run.jar.

Shell scripts to run the compiler

Basically just run runCompile.bash with a directory as argument to compile a program written in this language.

run.bash	A generic shell script to run the generated compiler. It needs the directory containing the program to compile as an argument. It permits the user to add additional arguments, such as -debug. It creates program.err, program.parse, program.print, usercode.user.s.
runCompile.bash	A more specific shell script, that calls run.bash, with the directory containing the program to compile as an argument. For example, type "runCompile.bash Programs/test".

Shell script to run the assembly language

Basically just run runSim.bash with a directory as argument to run the assembly language program generated by the compiler.

runSim.bash	Run the Alpha simulator in batch mode. It needs the directory containing the program to compile as an argument. It creates sim.err and sim.out. For example, type "runSim.bash Programs/test". Really intended for use by the markers. It is generally better for you to run the simulator interactively, until you believe your code is working.

Shell scripts to run the compiler or assembly language over all subdirectories of a directory.

Once everything seems to be close to working, use runallCompile.bash to compile all programs in this language, and runallSim.bash to run all the generated assembly language programs.

runall.bash	A general shell script to run another shell script over all subdirectories of the argument directory. Do not use directly.
runallCompile.bash	Runs runCompile.bash for every subdirectory of the argument directory. For example, type "runallCompile.bash Programs".
runallSim.bash	Runs runSim.bash for every subdirectory of the argument directory. For example, type "runallSim.bash Programs".

Notes:

Read the provided code before starting.

The code for "true" and "false" is very similar to the code for integer literals.

For example, the statement

```
a = true;
```

should generate code

```
{
     ldiq $t0, a;
     ldiq $t1, 1;
     stq $t1, ($t0);
}
```

The "||" and "&&" operators are trivial. Just define precedence as PREC_OR, PREC_AND; operator as "||", "&&"; and opCode as "or", "and". To generate code for ">", ">=", "!=", generate code as for "<=", "<", "==", then generate code to complement the result by generating a "cmpeq" instruction that compares freeReg with "0", and puts the result back in freeReg. To generate code for "!", generate code for the operand, then generate a "cmpeq" instruction, as for ">", ">=", "!=".

For example, the statement

```
cond = a < b || c > d;
```

should generate code

```
{
     ldiq $t0, cond;
     ldiq $t1, a;
     ldq $t1, ($t1);
     ldiq $t2, b;
     ldq $t2, ($t2);
     cmplt $t1, $t2;
     ldiq $t2, c;
     ldq $t2, ($t2);
     ldiq $t3, d;
```

```
            ldq $t3, ($t3);
            cmple $t2, $t3;
            cmpeq $t2, 0;
            or $t1, $t2;
            stq $t1, ($t0);
        }
```

The statements

```
a = 3;
b = 4;
print( "%d < %d is %d\n", a, b, a < b );
print( "%d <= %d is %d\n", a, b, a <= b );
print( "%d > %d is %d\n", a, b, a > b );
print( "%d >= %d is %d\n", a, b, a >= b );
print( "%d == %d is %d\n", a, b, a == b );
print( "%d != %d is %d\n", a, b, a != b );
a = true;
b = false;
print( "%d || %d is %d\n", a, b, a || b );
print( "%d && %d is %d\n", a, b, a && b );
print( "! %d is %d\n", a, ! a );
```

should generate output

```
3 < 4 is 1
3 <= 4 is 1
3 > 4 is 0
3 >= 4 is 0
3 == 4 is 0
3 != 4 is 1
1 || 0 is 1
1 && 0 is 0
! 1 is 0
```

The code for “if-then”, “if-then-else”, and “for” statements is similar to the code for “while” statements.

For example, the statements

```
a = 3;
b = 4;
if a < b then
     print( "%d < %d\n", a, b );
if a == b then
     print( "%d == %d\n", a, b );
if a > b then
     print( "%d > %d\n", a, b );
```

should generate output

```
3 < 4
```

The statements

```
for i = 0 to 5 do
     if i / 2 * 2 == i then
          print( "%d is even\n", i );
     else
          print( "%d is odd\n", i );
```

should generate output

```
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

Support methods are declared in the Code class to generate code for instructions, labels, etc. The text generated contains "%+", "%-", "%n" directives to increment and decrement the indenting level, generate line breaks, etc. This is then processed by the code for the FormattedOutput.print() method, to generated appropriately indented text.

Ensure that your code assembles and executes correctly, by loading it into the alpha simulator and executing it. You can also run the Alpha simulator in batch mode by using the runSim.bash shell script.

Submission:

Note that template files are provided for you to edit. Do not create your own files.

Run the command

```
createtar.bash
```

to create a tar/gzip file

```
ASSIGN1.tar.gz
```

containing

(a) An Excel file, "lalrParse.xls", with 2 separate sheets containing

  (i) The LALR(1) action/goto tables.

  (ii) The LALR(1) parse of the input.

(b) A Visio file "lalrDiagram.vsd", with 2 separate sheets containing

  (i) A drawing of the parse tree for the input.

  (ii) A drawing of the abstract syntax tree for the input.

(c) The COMPILE directory containing your solution for question 2.

Submit the tar/gzip file

```
ASSIGN1.tar.gz
```

using the assignment drop box.

Due Date 12 Noon, Wednesday (week 5) 28th March 2007, subject to normal Bonus/Penalty rules, as described in the introductory handout.

Appendix 1 CUP grammar for question 1

```
terminal
     LEXERROR,
     DOT, EXPANDSTO,    COMMA,    LEFT,     RIGHT,    LEFTSQ,   RIGHTSQ,  BAR;
//   .    :-            ,         (         )         [         ]         |

terminal String INTVALUE;    //   [0-9]+
terminal String NAME;        //   [a-z][A-Za-z0-9]*
terminal String VARIABLE;    //   [A-Z][A-Za-z0-9]*

nonterminal ProgramNode
     Program;
nonterminal ClauseListNode
     ClauseList;
nonterminal ClauseNode
     Clause;
nonterminal StructureListNode
     StructureList;
nonterminal StructureNode
     Structure;
nonterminal ExprListNode
     ExprList;
nonterminal ExprNode
     Expr, List, ElementListOpt, ElementList;

start with Program;

Program::=
          ClauseList:clauseList
          {:
          RESULT = new ProgramNode( clauseList );
          :}
     ;

ClauseList::=
          ClauseList:clauseList Clause:clause
          {:
          clauseList.addElement( clause );
          RESULT = clauseList;
          :}
     |
          Clause:clause
          {:
          RESULT = new ClauseListNode( clause );
          :}
     ;

Clause::=
          Structure:head DOT
          {:
          RESULT = new FactNode( head );
          :}
     |
          Structure:head EXPANDSTO StructureList:tail DOT
          {:
          RESULT = new RuleNode( head, tail );
          :}
     |
          error DOT
```

```
          {:
          RESULT = new ErrorClauseNode();
          :}
     ;

StructureList::=
          Structure:structure
          {:
          RESULT = new StructureListNode( structure );
          :}
     |
          StructureList:structureList COMMA Structure:structure
          {:
          structureList.addElement( structure );
          RESULT = structureList;
          :}
     ;

Structure::=
          NAME:name LEFT ExprList:exprList RIGHT
          {:
          RESULT = new StructureNode( name, exprList );
          :}
     ;

ExprList::=
          Expr:expr
          {:
          RESULT = new ExprListNode( expr );
          :}
     |
          ExprList:exprList COMMA Expr:expr
          {:
          exprList.addElement( expr );
          RESULT = exprList;
          :}
     ;

Expr::=
          INTVALUE:value
          {:
          RESULT = new IntValueNode( new Integer( value ).intValue() );
          :}
     |
          NAME:name
          {:
          RESULT = new NameNode( name );
          :}
     |
          VARIABLE:name
          {:
          RESULT = new VariableNode( name );
          :}
     |
          Structure:structure
          {:
          RESULT = structure;
          :}
     |
          List:list
          {:
```

```
          RESULT = list;
          :}
     ;

List::=
          LEFTSQ ElementListOpt:elementList RIGHTSQ
          {:
          RESULT = elementList;
          :}
     ;

ElementListOpt::=
          ElementList:elementList
          {:
          RESULT = elementList;
          :}
     |
          /* Empty */
          {:
          RESULT = new EmptyListNode();
          :}
     ;

ElementList::=
          Expr:expr
          {:
          RESULT = new NonEmptyListNode( expr, new EmptyListNode() );
          :}
     |
          Expr:expr COMMA ElementList:elementList
          {:
          RESULT = new NonEmptyListNode( expr, elementList );
          :}
     |
          Expr:expr BAR Expr:tail
          {:
          RESULT = new NonEmptyListNode( expr, tail );
          :}
     ;
```

Appendix 2 LALR(1) State Information

```
===== Rules =====
[0] $START ::= Program EOF
[1] Program ::= ClauseList
[2] ClauseList ::= ClauseList Clause
[3] ClauseList ::= Clause
[4] Clause ::= Structure DOT
[5] Clause ::= Structure EXPANDSTO StructureList DOT
[6] Clause ::= error DOT
[7] StructureList ::= Structure
[8] StructureList ::= StructureList COMMA Structure
[9] Structure ::= NAME LEFT ExprList RIGHT
[10] ExprList ::= Expr
[11] ExprList ::= ExprList COMMA Expr
[12] Expr ::= INTVALUE
[13] Expr ::= NAME
[14] Expr ::= VARIABLE
[15] Expr ::= Structure
[16] Expr ::= List
[17] List ::= LEFTSQ ElementListOpt RIGHTSQ
```

```
[18] ElementListOpt ::= ElementList
[19] ElementListOpt ::=
[20] ElementList ::= Expr
[21] ElementList ::= Expr COMMA ElementList
[22] ElementList ::= Expr BAR Expr

-------- ACTION_TABLE --------
From state #0
     error:SHIFT(state 4) NAME:SHIFT(state 6)
From state #1
     EOF:SHIFT(state 36)
From state #2
     DOT:SHIFT(state 30) EXPANDSTO:SHIFT(state 29)
From state #3
     EOF:REDUCE(rule 1) error:SHIFT(state 4) NAME:SHIFT(state 6)
From state #4
     DOT:SHIFT(state 27)
From state #5
     EOF:REDUCE(rule 3) error:REDUCE(rule 3) NAME:REDUCE(rule 3)
From state #6
     LEFT:SHIFT(state 7)
From state #7
     LEFTSQ:SHIFT(state 15) INTVALUE:SHIFT(state 12) NAME:SHIFT(state 13)
     VARIABLE:SHIFT(state 14)
From state #8
     COMMA:REDUCE(rule 15) RIGHT:REDUCE(rule 15) RIGHTSQ:REDUCE(rule 15)
     BAR:REDUCE(rule 15)
From state #9
     COMMA:SHIFT(state 24) RIGHT:SHIFT(state 25)
From state #10
     COMMA:REDUCE(rule 16) RIGHT:REDUCE(rule 16) RIGHTSQ:REDUCE(rule 16)
     BAR:REDUCE(rule 16)
From state #11
     COMMA:REDUCE(rule 10) RIGHT:REDUCE(rule 10)
From state #12
     COMMA:REDUCE(rule 12) RIGHT:REDUCE(rule 12) RIGHTSQ:REDUCE(rule 12)
     BAR:REDUCE(rule 12)
From state #13
     COMMA:REDUCE(rule 13) LEFT:SHIFT(state 7) RIGHT:REDUCE(rule 13)
     RIGHTSQ:REDUCE(rule 13) BAR:REDUCE(rule 13)
From state #14
     COMMA:REDUCE(rule 14) RIGHT:REDUCE(rule 14) RIGHTSQ:REDUCE(rule 14)
     BAR:REDUCE(rule 14)
From state #15
     LEFTSQ:SHIFT(state 15) RIGHTSQ:REDUCE(rule 19) INTVALUE:SHIFT(state 12)
     NAME:SHIFT(state 13) VARIABLE:SHIFT(state 14)
From state #16
     RIGHTSQ:SHIFT(state 23)
From state #17
     COMMA:SHIFT(state 19) RIGHTSQ:REDUCE(rule 20) BAR:SHIFT(state 20)
From state #18
     RIGHTSQ:REDUCE(rule 18)
From state #19
     LEFTSQ:SHIFT(state 15) INTVALUE:SHIFT(state 12) NAME:SHIFT(state 13)
     VARIABLE:SHIFT(state 14)
From state #20
     LEFTSQ:SHIFT(state 15) INTVALUE:SHIFT(state 12) NAME:SHIFT(state 13)
     VARIABLE:SHIFT(state 14)
From state #21
     RIGHTSQ:REDUCE(rule 22)
From state #22
```

```
     RIGHTSQ:REDUCE(rule 21)
From state #23
     COMMA:REDUCE(rule 17) RIGHT:REDUCE(rule 17) RIGHTSQ:REDUCE(rule 17)
     BAR:REDUCE(rule 17)
From state #24
     LEFTSQ:SHIFT(state 15) INTVALUE:SHIFT(state 12) NAME:SHIFT(state 13)
     VARIABLE:SHIFT(state 14)
From state #25
     DOT:REDUCE(rule 9) EXPANDSTO:REDUCE(rule 9) COMMA:REDUCE(rule 9)
     RIGHT:REDUCE(rule 9) RIGHTSQ:REDUCE(rule 9) BAR:REDUCE(rule 9)
From state #26
     COMMA:REDUCE(rule 11) RIGHT:REDUCE(rule 11)
From state #27
     EOF:REDUCE(rule 6) error:REDUCE(rule 6) NAME:REDUCE(rule 6)
From state #28
     EOF:REDUCE(rule 2) error:REDUCE(rule 2) NAME:REDUCE(rule 2)
From state #29
     NAME:SHIFT(state 6)
From state #30
     EOF:REDUCE(rule 4) error:REDUCE(rule 4) NAME:REDUCE(rule 4)
From state #31
     DOT:REDUCE(rule 7) COMMA:REDUCE(rule 7)
From state #32
     DOT:SHIFT(state 34) COMMA:SHIFT(state 33)
From state #33
     NAME:SHIFT(state 6)
From state #34
     EOF:REDUCE(rule 5) error:REDUCE(rule 5) NAME:REDUCE(rule 5)
From state #35
     DOT:REDUCE(rule 8) COMMA:REDUCE(rule 8)
From state #36
     EOF:REDUCE(rule 0)
------------------------------
-------- REDUCE_TABLE --------
From state #0:
     Program:GOTO(1)
     ClauseList:GOTO(3)
     Clause:GOTO(5)
     Structure:GOTO(2)
From state #1:
From state #2:
From state #3:
     Clause:GOTO(28)
     Structure:GOTO(2)
From state #4:
From state #5:
From state #6:
From state #7:
     Structure:GOTO(8)
     ExprList:GOTO(9)
     Expr:GOTO(11)
     List:GOTO(10)
From state #8:
From state #9:
From state #10:
From state #11:
From state #12:
From state #13:
From state #14:
From state #15:
     Structure:GOTO(8)
```

```
     Expr:GOTO(17)
     List:GOTO(10)
     ElementListOpt:GOTO(16)
     ElementList:GOTO(18)
From state #16:
From state #17:
From state #18:
From state #19:
     Structure:GOTO(8)
     Expr:GOTO(17)
     List:GOTO(10)
     ElementList:GOTO(22)
From state #20:
     Structure:GOTO(8)
     Expr:GOTO(21)
     List:GOTO(10)
From state #21:
From state #22:
From state #23:
From state #24:
     Structure:GOTO(8)
     Expr:GOTO(26)
     List:GOTO(10)
From state #25:
From state #26:
From state #27:
From state #28:
From state #29:
     StructureList:GOTO(32)
     Structure:GOTO(31)
From state #30:
From state #31:
From state #32:
From state #33:
     Structure:GOTO(35)
From state #34:
From state #35:
From state #36:
------------------------------
```

Appendix 3 Input for question 1(b), (c), (d)

```
last( [ X, Y | Z ], W ):- last( [ Y | Z ], W ).
```

Appendix 4 Provided grammar for question 2

```
terminal
     LEXERROR,
     LT, LE, EQ, PLUS, MINUS, TIMES, DIVIDE, ASSIGN,
     PRINT, IF, THEN, ELSE, WHILE, DO, FOR, TO, SEMICOLON, LEFTCURLY, RIGHTCURLY,
     LEFT, RIGHT, COMMA;
terminal String STRINGVALUE;
terminal String INTVALUE;
terminal String IDENT;

nonterminal ProgramNode
     Program;

nonterminal StmtListNode
     StmtList;
```

```
nonterminal StmtNode
     Stmt;

nonterminal ExprListNode
     ExprList;

nonterminal ExprNode
     Expr, RelExpr, PlusExpr, MulExpr, Primary;

start with Program;

Program::=
          StmtList:stmtList
          {:
          RESULT = new ProgramNode( stmtList );
          :}
     ;

StmtList::=
          {:
          RESULT = new StmtListNode();
          :}
     |
          StmtList:stmtList Stmt:stmt
          {:
          stmtList.addElement( stmt );
          RESULT = stmtList;
          :}
     ;

Stmt::=
          IDENT:ident ASSIGN Expr:expr SEMICOLON
          {:
          RESULT = new AssignStmtNode( ident, expr );
          :}
     |
          PRINT LEFT ExprList:exprList RIGHT SEMICOLON
          {:
          RESULT = new PrintStmtNode( exprList );
          :}
     |
          WHILE Expr:expr DO Stmt:stmt
          {:
          RESULT = new WhileStmtNode( expr, stmt );
          :}
     |
          LEFTCURLY StmtList:stmtList RIGHTCURLY
          {:
          RESULT = new CompoundStmtNode( stmtList );
          :}
     |
          error SEMICOLON
          {:
          RESULT = new ErrorStmtNode();
          :}
     |
          error RIGHTCURLY
          {:
          RESULT = new ErrorStmtNode();
          :}
     ;
```

```
ExprList::=
          Expr:expr
          {:
          RESULT = new ExprListNode( expr );
          :}
     |
          ExprList:exprList COMMA Expr:expr
          {:
          exprList.addElement( expr );
          RESULT = exprList;
          :}
     ;

Expr::=
          RelExpr:expr
          {:
          RESULT = expr;
          :}
     ;

RelExpr::=
          PlusExpr:expr1 LT PlusExpr:expr2
          {:
          RESULT = new LessThanNode( expr1, expr2 );
          :}
     |
          PlusExpr:expr1 LE PlusExpr:expr2
          {:
          RESULT = new LessEqualNode( expr1, expr2 );
          :}
     |
          PlusExpr:expr1 EQ PlusExpr:expr2
          {:
          RESULT = new EqualNode( expr1, expr2 );
          :}
     |
          PlusExpr:expr
          {:
          RESULT = expr;
          :}
     ;

PlusExpr::=
          PlusExpr:expr PLUS MulExpr:expr2
          {:
          RESULT = new PlusNode( expr, expr2 );
          :}
     |
          PlusExpr:expr MINUS MulExpr:expr2
          {:
          RESULT = new MinusNode( expr, expr2 );
          :}
     |
          MINUS MulExpr:expr2
          {:
          RESULT = new NegateNode( expr2 );
          :}
     |
          MulExpr:expr
          {:
```

```
          RESULT = expr;
          :}
     ;

MulExpr::=
          MulExpr:expr1 TIMES Primary:expr2
          {:
          RESULT = new TimesNode( expr1, expr2 );
          :}
     |
          MulExpr:expr1 DIVIDE Primary:expr2
          {:
          RESULT = new DivideNode( expr1, expr2 );
          :}
     |
          Primary:expr
          {:
          RESULT = expr;
          :}
     ;

Primary::=
          LEFT Expr:expr RIGHT
          {:
          RESULT = expr;
          :}
     |
          INTVALUE:value
          {:
          RESULT = new IntValueNode( Integer.parseInt( value ) );
          :}
     |
          STRINGVALUE:value
          {:
          RESULT = new StringValueNode(
               Convert.parseString( value.substring( 1, value.length() - 1 ) ) );
          :}
     |
          IDENT:ident
          {:
          RESULT = new IdentNode( ident );
          :}
     ;
```

Appendix 5 Sample program and code generated for question 2

The program

```
max = 20;
for p = 2 to max do
     {
          isPrime = true;
          i = 2;
          while i * i < p do
               {
                    if p / i * i == p then
                         isPrime = false;
                    i = i + 1;
               }
          if isPrime then
```

```
            print( "%d is prime\n", p );
    }
```

might generate code

```
entry main.enter;
import "../../IMPORT/callsys.h";
import "../../IMPORT/proc.h";
import "../../IMPORT/callsys.lib.s";
import "../../IMPORT/string.lib.s";
import "../../IMPORT/number.lib.s";
import "../../IMPORT/io.lib.s";
public block main {
    data {
    max:
        quad 0;
    p:
        quad 0;
    isPrime:
        quad 0;
    i:
        quad 0;
    } data
    code {
    public enter:
        {
            ldiq $t0, max;
            ldiq $t1, 20;
            stq $t1, ($t0);
        }
        {
        for:
            ldiq $t0, 2;
            ldiq $t1, p;
            stq $t0, ($t1);
        while:
            ldiq $t0, p;
            ldq $t0, ($t0);
            ldiq $t1, max;
            ldq $t1, ($t1);
            cmple $t0, $t1;
            blbc $t0, end;
        do:
            {
                {
                    ldiq $t0, isPrime;
                    ldiq $t1, 1;
                    stq $t1, ($t0);
                }
                {
                    ldiq $t0, i;
                    ldiq $t1, 2;
                    stq $t1, ($t0);
                }
                {
                while:
                    ldiq $t0, i;
                    ldq $t0, ($t0);
                    ldiq $t1, i;
                    ldq $t1, ($t1);
                    mulq $t0, $t1;
                    ldiq $t1, p;
                    ldq $t1, ($t1);
```

```
     cmplt $t0, $t1;
     blbc $t0, end;
do:
     {
          {
          if:
               ldiq $t0, p;
               ldq $t0, ($t0);
               ldiq $t1, i;
               ldq $t1, ($t1);
               divq $t0, $t1;
               ldiq $t1, i;
               ldq $t1, ($t1);
               mulq $t0, $t1;
               ldiq $t1, p;
               ldq $t1, ($t1);
               cmpeq $t0, $t1;
               blbc $t0, end;
          then:
               {
                    ldiq $t0, isPrime;
                    ldiq $t1, 0;
                    stq $t1, ($t0);
               }
          end:
          }
          {
               ldiq $t0, i;
               ldiq $t1, i;
               ldq $t1, ($t1);
               ldiq $t2, 1;
               addq $t1, $t2;
               stq $t1, ($t0);
          }
     }
     br while;
end:
}
{
if:
     ldiq $t0, isPrime;
     ldq $t0, ($t0);
     blbc $t0, end;
then:
     {
          {
               const {
               string:
                    asciiz "%d is prime\n";
                    align;
               } const
               ldiq $t0, string;
          }
          mov $t0, $a0;
          ldiq $t0, p;
          ldq $t0, ($t0);
          mov $t0, $a1;
          bsr IO.printf.enter;
     }
end:
}
```

```
            }
        increment:
            ldiq $t0, p;
            ldq $t1, ($t0);
            addq $t1, 1;
            stq $t1, ($t0);
            br while;
        end:
        }
        clr $a0;
        bsr Sys.exit.enter;
    } code
} block main
```

and output

```
2 is prime
3 is prime
4 is prime
5 is prime
7 is prime
9 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

Bruce Hutton