COP5621 - Spring 2020
Assignment 3
cparse

This assignment consists of two parts. First, you will write a program to read from standard input an augmented grammar, follow information for that grammar, and a collection of sets of LR(0) items. This program will use this input to produce the action and goto table information along with the appropriate defined constants. You should call the file containing the first program *gentable.c*. Second, you will write a program that will include the file containing the table information, parse the input using the SLR parsing algorithm, and produce a list of the reductions that have occurred. You should call the file containing the second program *cparse.c*.

The input for *gentable* consists of (1) an augmented grammar, (2) follow information, and (3) a collection of sets of LR(0) items. An example input file is given below and on the following page and is available in the *items.txt* file in the *~whalley/cop5621ex* directory. Each production in the augmented grammar has a nonterminal (capital letter) on the left hand side, an arrow (->), and a right hand side that consists of nonterminals and terminals (single characters that are not capital letters, are not the "@", "$", or "'" symbols, and are not a blank, tab, or newline). Note that the first production will have a "'" on the left hand side to represent the augmented grammar's start symbol.

The follow information indicates what tokens can follow each nonterminal in a sentential form. Each line consists of a nonterminal and the set of terminal symbols that comprise the follow of that nonterminal. The "$" symbol represents the end of input.

Each set of LR(0) items consists of an identier number (I%d:) followed by the items within the set. Each item is a production in the set and includes the cursor ("@") to indicate the current parsing position. The first item in the set where the cursor precedes a particular grammar symbol also includes a goto specification indicating the next set (state).

```
Augmented Grammar
-----------------
´->E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i

Follows
-------
E +)$
T *+)$
F *+)$

Sets of LR(0) Items
-------------------
I0:
   '->@E                goto(E)=I1
   E->@E+T
   E->@T                goto(T)=I2
   T->@T*F
   T->@F                goto(F)=I3
   F->@(E)              goto(()=I4
   F->@i                goto(i)=I5
```

```
I1:
   ’->E@
   E->E@+T              goto(+)=I6

I2:
   E->T@
   T->T@*F              goto(*)=I7

I3:
   T->F@

I4:
   F->(@E)              goto(E)=I8
   E->@E+T
   E->@T                goto(T)=I2
   T->@T*F
   T->@F                goto(F)=I3
   F->@(E)              goto(()=I4
   F->@i                goto(i)=I5

I5:
   F->i@

I6:
   E->E+@T              goto(T)=I9
   T->@T*F
   T->@F                goto(F)=I3
   F->@(E)              goto(()=I4
   F->@i                goto(i)=I5

I7:
   T->T*@F              goto(F)=I10
   F->@(E)              goto(()=I4
   F->@i                goto(i)=I5

I8:
   F->(E@)              goto())=I11
   E->E@+T              goto(+)=I6

I9:
   E->E+T@
   T->T@*F              goto(*)=I7

I10:
   T->T*F@

I11:
   F->(E)@
```

You can assume that the maximum number of productions will be 50, the maximum number of sets will be 100, the maximum number of terminal symbols will be 50, the maximum number of nonterminals will be 50, and the maximum number of characters in a production will be 50.

The *gentable* program should write to standard output the data that represents the action and goto information along with appropriate constants and other information that is used to parse input associated with the specified grammar. The output of *gentable* given the input from the previous page is shown below and on the following page and is available in the *tables.h* file in the *~whalley/cop5621ex* directory. You should make the output of your *gentable* solution match my output exactly.

```
#define NUM_STATES   12
#define NUM_TERMS    6
#define NUM_NONTERMS 3
#define NUM_PRODS    7

static char action[NUM_STATES][NUM_TERMS] = {
 /*   +    *    (    )    i    $   */
   { 'e', 'e', 's', 'e', 's', 'e' }, /*  0 */
   { 's', 'e', 'e', 'e', 'e', 'a' }, /*  1 */
   { 'r', 's', 'e', 'r', 'e', 'r' }, /*  2 */
   { 'r', 'r', 'e', 'r', 'e', 'r' }, /*  3 */
   { 'e', 'e', 's', 'e', 's', 'e' }, /*  4 */
   { 'r', 'r', 'e', 'r', 'e', 'r' }, /*  5 */
   { 'e', 'e', 's', 'e', 's', 'e' }, /*  6 */
   { 'e', 'e', 's', 'e', 's', 'e' }, /*  7 */
   { 's', 'e', 'e', 's', 'e', 'e' }, /*  8 */
   { 'r', 's', 'e', 'r', 'e', 'r' }, /*  9 */
   { 'r', 'r', 'e', 'r', 'e', 'r' }, /* 10 */
   { 'r', 'r', 'e', 'r', 'e', 'r' }  /* 11 */
};

static int action_num[NUM_STATES][NUM_TERMS] = {
 /*   +   *   (   )   i   $  */
   {  0,  0,  4,  0,  5,  0 }, /*  0 */
   {  6,  0,  0,  0,  0,  0 }, /*  1 */
   {  2,  7,  0,  2,  0,  2 }, /*  2 */
   {  4,  4,  0,  4,  0,  4 }, /*  3 */
   {  0,  0,  4,  0,  5,  0 }, /*  4 */
   {  6,  6,  0,  6,  0,  6 }, /*  5 */
   {  0,  0,  4,  0,  5,  0 }, /*  6 */
   {  0,  0,  4,  0,  5,  0 }, /*  7 */
   {  6,  0,  0, 11,  0,  0 }, /*  8 */
   {  1,  7,  0,  1,  0,  1 }, /*  9 */
   {  3,  3,  0,  3,  0,  3 }, /* 10 */
   {  5,  5,  0,  5,  0,  5 }  /* 11 */
};

static int go_to[NUM_STATES][NUM_NONTERMS] = {
 /*   E   T   F  */
   {  1,  2,  3 }, /*  0 */
   {  0,  0,  0 }, /*  1 */
   {  0,  0,  0 }, /*  2 */
   {  0,  0,  0 }, /*  3 */
   {  8,  2,  3 }, /*  4 */
   {  0,  0,  0 }, /*  5 */
   {  0,  9,  3 }, /*  6 */
   {  0,  0, 10 }, /*  7 */
   {  0,  0,  0 }, /*  8 */
   {  0,  0,  0 }, /*  9 */
   {  0,  0,  0 }, /* 10 */
   {  0,  0,  0 }  /* 11 */
};
```

```
static int reduce_num[NUM_PRODS] =
 /*  1  2  3  4  5  6  7 */
   { 1, 3, 1, 3, 1, 3, 1 };

static int reduce_lhs[NUM_PRODS] =
 /*  1  2  3  4  5  6  7 */
   { 0, 0, 0, 1, 1, 2, 2 };

static char tokens[NUM_TERMS] =
 /*   0    1    2    3    4    5  */
   { '+', '*', '(', ')', 'i', '$' };
```

The *cparse* program will include the file produced by *gentable*. You should include it as the file *tables.h*. This program will read from standard input and parse the input using the SLR parsing algorithm discussed in class. You should write a lexical analyzer that skips whitespace and returns the next token. You should use the `tokens` array in the *tables.h* file to determine the next token to return. Remember that each token is a single character. You can assume that the maximum number of states on the stack during the parse is 100. You should use the `action`, `action_num`, and `go_to` to determine the appropriate action to perform. Whenever a reduction occurs, you should print out:

```
"reduce #"
```

where # is the production number, to standard output. You should use the `reduce_num` and `reduce_lhs` arrays to update the stack on a reduce operation. At the end of the execution you should print one of the following lines depending upon the result of the parse.

```
"Accept state reached.",
"Stack overflow occurred."
"Error state on token '&' at state #."
```

where & is the token character and # is the state number. For instance, given the following input to the *cparse* program:

```
i*i + i
```

along with the *tables.h* file shown on the previous page, the output should be as shown below.

```
reduce 6
reduce 4
reduce 6
reduce 3
reduce 2
reduce 6
reduce 4
reduce 1

Accept state reached.
```

E-mail your *gentable.c* and *cparse.c* source files as attachments to *whalley@cs.fsu.edu* before the beginning of class on February 18. Please put COP5621 somewhere on the subject line of the message.