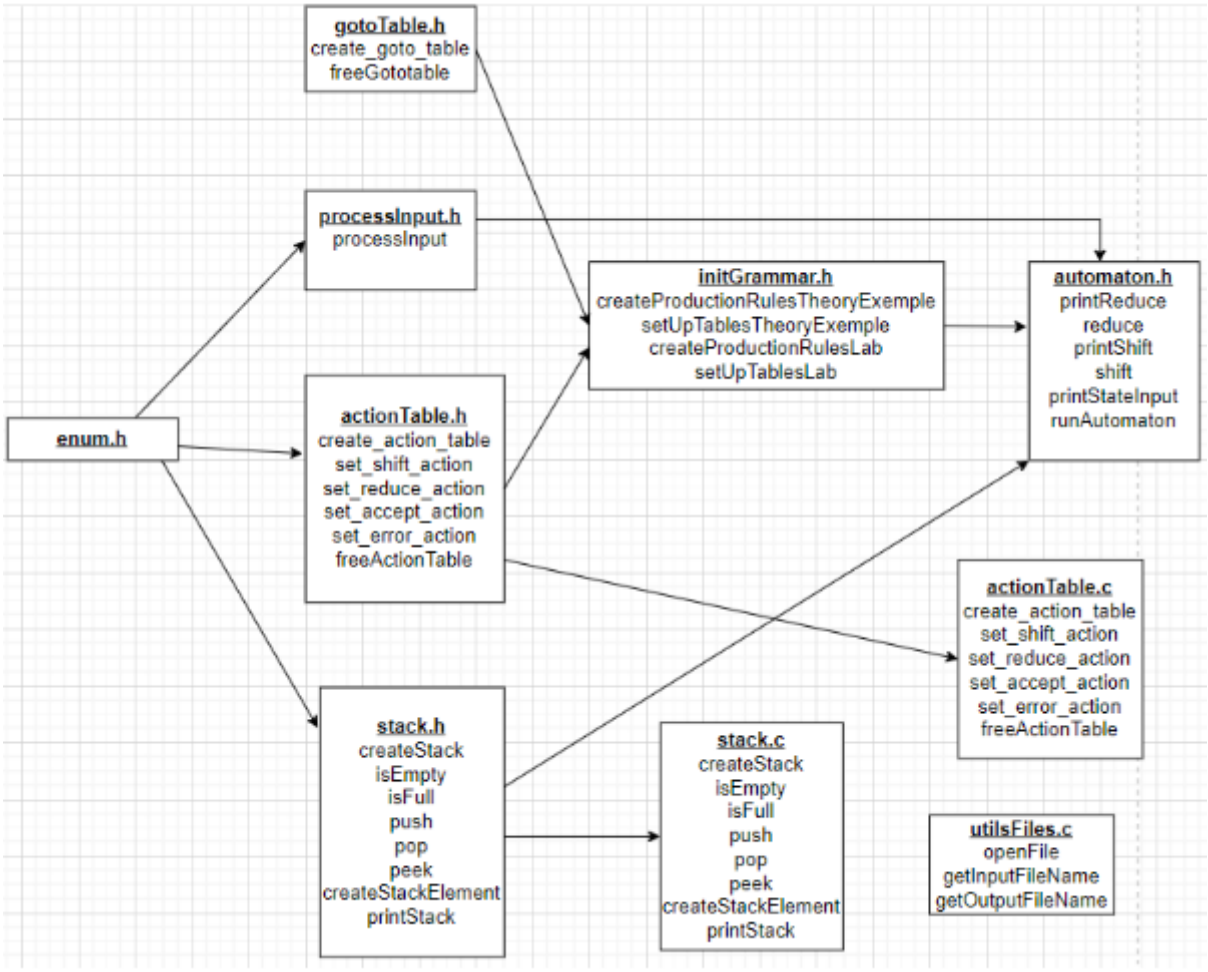
DESIGN AND DOCUMENTATION LAB 3

DESIGN AND DOCUMENTATION LAB 3

Explanation of the diagram:



- `enum.h`: Grammar symbols are defined, the actions that the automaton can perform, and where we can switch between running the theory grammar or the lab grammar.
- `Stack.C`: Classic stack structure, used to execute the shift/reduce functions of the automaton.
- `actionTable.c`: implementation of the action table, the table is created with the `create_action_table` function, and the actions are set using the `set_shift_action`, `set_reduce_action`, `set_accept_action and set_error_action` functions. Each cell in the table represents a state and a symbol, and the action structure determines which action to take based on that state and symbol combination. Actions can be of four types: shift, reduce, accept and error. The memory allocated to the table is freed with the `freeActionTable` function.
- `processInput.h`: Contains a function named `processInput` that reads an input file and extracts tokens from it. It takes in a file pointer and an array of tokens as arguments and returns an integer

representing the number of tokens in the input file. The function uses string functions to extract the lexeme and type of each token and assigns an index based on its type or lexeme. It appends a special end-of-file token to the array of tokens.

- `gotoTable.h`: contains two functions related to a goto table. The first function, named `create_goto_table`, allocates memory for the goto table and initializes it. It takes in two integers representing the number of states and symbols in the table and returns a pointer to the allocated memory for the goto table. The second function, named `freeGotoTable`, frees the memory allocated for the goto table. It takes in the goto table and the number of states in the table and returns void. The memory allocated for each row in the table is freed before freeing the memory allocated for the table itself.
- `initGrammar.h`: This file contains four functions, `createProductionRulesTheoryExemple` and `setUpTablesTheoryExemple`, and the same for the lab example, and two preprocessor directives for different cases. If the preprocessor directive `CASE` is defined as `THEORY`, the first function creates an array of production rules for a theoretical example and returns a pointer to this array. The second function sets up the action and goto tables for the theoretical example, using pointers to pointers to Action and int arrays, respectively. If the preprocessor directive `CASE` is defined as `LAB`, the first function creates an array of production rules for a lab example and returns a pointer to this array. The second function sets up the action and goto tables for the lab example, using pointers to pointers to Action and int arrays, respectively. The details of the implementation are given in the comments of the code.
- `automaton.h`: This file contains several functions related to running an LR parser for a given input
  - The `printReduce` function writes information about reducing a production rule and updating the stack to an output file.
  - The `reduce function reduces the stack according to a given production rule and updates the stack accordingly.
  - The `printShift` function writes information about shifting a token and updating the stack to an output file.
  - The `shift` function shifts a token onto the stack and updates the stack accordingly.
  - The `printStateInput` function writes information about the current state and current input token to an output file.
  - The `runAutomaton`function runs the LR parser for the given input tokens and writes the actions taken to the output file. It uses the other functions in this file and takes in production rules, action and goto tables, tokens, and an output file as input.
- `utilsFiles.c`: This file contains functions declarations for file handling. It provides functions to open a file with a given name, get an input file name, and build an output file name by adding "p3dbg.txt" at the end of the input file name.

Data structures:

Action:

```
typedef struct
{
    enum ActionType type;
    int state_or_production;
} Action;
```

- This structure “Action” has two fields: “type” which means the type of action that can be “shift”, “reduce”, “accept” and “error” and “state_or_production” is an integer value that represents either a state or a production.

production_rule:

```
typedef struct production_rule
{
    char lhs[1];
    int lhs_index;
    char rhs[10];
    int rhs_len;
} production_rule;
```

- This structure “production_rule” contains four fields: “lhs” which represents the left-hand side of the production rule, “lhs_index” an integer value that represents the index of the nonterminal symbol on the left-hand side of the production rule, “rhs” which represents the right-hand side of the production rule and “rhs_len” an integer value that represents the length of the right-hand side of the production rule.

StackElement:

```
typedef struct
{
    char symbol[20];
    int state;
} StackElement;
```

- This structure “StackElement” contains two fields: “symbol” an array of characters with length 20, which represents a symbol on the stack and “state” an integer value that represents the state associated with the symbol.

Stack:

```
typedef struct
{
    StackElement data[MAX_SIZE];
    int top;
} Stack;
```

- This structure “Stack” contains two fields: “data” an array of “StackElement” struct with length MAX_SIZE, which represents the underlying data structure of the stack and “top” an integer value that represents the index of the top element in the stack.

enum ActionType:

```
enum ActionType
{
    SHIFT,
    REDUCE,
    ACCEPT,
    ERROR
};
```

- There are 4 possible action types: “SHIFT” means that the parser should shift the next token onto the stack, “REDUCE” means that the parser should reduce a set of symbols on the stack to a non-terminal, “ACCEPT” means that the parser has successfully parsed the input and should accept it and “ERROR” means that the parser has encountered an error while parsing the input.

enum Symbol (Symbols for theory slides grammar):

```
enum Symbol
{
    S,
    E,
    PLUS,
    LEFT_PARENTHESIS,
    RIGHT_PARENTHESIS,
    NUM,
    ACC,
};
```

- There are 7 possible symbols: “S” represents the start symbol of the grammar, “E” represents a non-terminal symbol in the grammar, “PLUS” represents the plus operator, “LEFT_PARENTHESIS” represents a left parenthesis, “RIGHT_PARENTHESIS” represents a right parenthesis, “NUM”

represents a numerical value in the grammar and “ACC” represents the accept symbol, which is used to indicate that the input has been successfully parsed.

enum Symbol_lab (Symbols for lab grammar):

- there are 10 possible symbols which are the same as the previous one but adding “T”, “F” and “MULT”. “T” and “F” are non-terminal symbols in the grammar and “MULT” represents the multiplication operator.

```
enum Symbol_lab
{
    S,
    E,
    T,
    F,
    PLUS,
    MULT,
    LEFT_PARENTHESIS,
    RIGHT_PARENTHESIS,
    NUM,
    ACC,
};
```

Comment for each function:

actionTable.c :

- `Action **create_action_table(int num_state, int num_symbol)`

```
    /**
     * @brief Create a new action table with the given number of states and
symbols
     * @param num_state (int) the number of states in the action table
     * @param num_symbol (int) the number of symbols in the action table
     * @return Action** a pointer to the new action table
     */
```

- `void set_shift_action(Action **action_table, int state, int symbol, int next_state)`

```
    /**
     * @brief Set a shift action in the action table
     * @param action_table (Action**) the action table to set the action in
     * @param state (int) the state to set the action in
     * @param symbol (int) the symbol to set the action for
     * @param next_state (int) the next state to shift to
     * @return void
     */
```

- `void set_reduce_action(Action **action_table, int state, int symbol, int production)`

```
    /**
     * @brief Set a reduce action in the action table
     * @param action_table (Action**) the action table to set the action in
     * @param state (int) the state to set the action in
     * @param symbol (int) the symbol to set the action for
     * @param production (int) the production to reduce to
     * @return void
     */
```

- `void set_accept_action(Action **action_table, int state, int symbol)`

```
    /**
     * @brief Set an accept action in the action table
     * @param action_table (Action**) the action table to set the action in
     * @param state (int) the state to set the action in
     * @param symbol (int) the symbol to set the action for
     * @return void
     */
```

- `void set_error_action(Action **action_table, int state, int symbol)`

```
    /**
     * @brief Set an error action in the action table
     * @param action_table (Action**) the action table to set the action in
     * @param state (int) the state to set the action in
     * @param symbol (int) the symbol to set the action for
     * @return void
     */
```

- `void freeActionTable(Action **action_table, int num_state)`

```
    /**
     * @brief Free the memory allocated for the action table
     * @param action_table (Action**) the action table to free
```

```
     * @param num_state (int) the number of states in the action table
     * @return void
     */
```

gotoTable.c :

- `int **create_goto_table(int num_state, int num_symbol)`

```
    /**
     * @brief Allocate memory for the goto table and initialize it
     * @param num_state (int) the number of states in the goto table
     * @param num_symbol (int) the number of symbols in the goto table
     * @return int** a pointer to the allocated memory for the goto table
     */
```

- `void freeGotoTable(int **goto_table, int num_state)`

```
    /**
     * @brief Free the memory allocated for the goto table
     * @param goto_table (int**) the goto table to free
     * @param num_state (int) the number of states in the goto table
     * @return void
     */
```

automaton.c :

- `void printReduce(FILE *output_file, production_rule production, int new_state, Stack *stack)`

```
    /**
     * @brief Write the reduce action and the updated stack content in the
output file
     * @param output_file Output file where the information will be written
     * @param production Production rule used for reducing
     * @param new_state New state of the stack after reducing
     * @param stack Pointer to the stack being used
     */
```

- `void reduce(production_rule production, Stack *stack, int **goto_table, FILE *output_file)`

```
    /**
     * @brief Reduce the stack according to the production rule
     * @param production Production rule used for reducing
```

```
     * @param stack Pointer to the stack being used
     * @param goto_table GOTO table for the automaton being used
     * @param output_file Output file w<br><br><br>

## Comment for each function:here the information will be written
     */
```

- `void printShift(FILE *output_file, Stack *stack, int next_state)`

```
    /**
     * @brief Write the shift action and the updated stack content in the
output file
     * @param output_file Output file where the information will be written
     * @param stack Pointer to the stack being used
     * @param next_state Next state of the stack after shifting
     */
```

- `void shift(Stack *stack, char token_lexeme[], int next_state, FILE *output_file)`

```
    /**
     * @brief Shift the stack according to the token
     * @param stack Pointer to the stack being used
     * @param token_lexeme Lexeme of the token being shifted
     * @param next_state Next state of the stack after shifting
     * @param output_file Output file where the information will be written
     */
```

- `void printStateInput(FILE *output_file, int current_state, char token_lexeme[])`

```
    /**
     * @brief Write the current state and current input token in the output
file
     * @param output_file Output file where the information will be written
     * @param current_state Current state of the stack
     * @param token_lexeme Lexeme of the token being processed
     */
```

- `void runAutomaton(production_rule *productions, Action **action_table, int **goto_table, Token tokens[], int num_tokens, FILE *output_file)`

```
    /**
     * @brief Runs the automaton for the given input tokens and writes the
actions taken to the output file
     * @param productions Array of production rules for the grammar
     * @param action_table Table of actions for the LR parser
     * @param goto_table Table of gotos for the LR parser
```

```
     * @param tokens Array of tokens representing the input to be parsed
     * @param num_tokens Number of tokens in the input
     * @param output_file Output file where the actions taken will be
written
     */
```

initGrammar.c :

- `production_rule *createProductionRulesTheoryExemple()`

```
    /**
     * @brief Creates production rules for the Theory example.
     * @return A pointer to an array of production rules.
     */
```

- `void setUpTablesTheoryExemple(Action **action_table, int **goto_table)`

```
    /**
     * @brief Sets up the action and goto tables for the Theory example
     *
     * @param action_table action_table A pointer to a pointer to an Action
struct
     * @param goto_table A pointer to a pointer to an int array.
     */
```

- `production_rule *createProductionRulesLab()`

```
    /**
     * @brief Creates production rules for the Lab example.
     * @return A pointer to an array of production rules.
     */
```

- `void setUpTablesLab(Action **action_table, int **goto_table)`

```
    /**
     * @brief Sets up the action and goto tables for the Lab example.
     * @param action_table A pointer to a pointer to an Action struct.
     * @param goto_table A pointer to a pointer to an int array.
     */
```

processInput.c :

- `int processInput(FILE *input_file, Token *tokens)`

```
/**
 * @brief Read the input file and extract the tokens
 * @param input_file the file containing the sequence of tokens
 * @param tokens an array which is going to store the tokens
 * @return the number of tokens contained in the input file
 */
```

Stack.c :

- `Stack *createStack()`

```
/**
 * @brief Initialize the stack
 * @param s the stack to be initialized
 * @return void
 */
```

- `int isEmpty(Stack *s)`

```
/**
 * @brief Check if the stack is empty
 * @param s the stack to be checked
 * @return int 1 if the stack is empty, 0 otherwise
 */
```

- `int isFull(Stack *s)`

```
/**
 * @brief Check if the stack is full
 * @param s the stack to be checked
 * @return int 1 if the stack is full, 0 otherwise
 */
```

- `void push(Stack *s, StackElement *item)`

```
/**
 * @brief Add an item to the stack
 * @param s the stack to add the item to
 * @param item the item to be added to the stack
 * @return void
 */
```

- `StackElement pop(Stack *s)`

```
/**
 * @brief Remove and return the top item from the stack
```

```
     * @param s the stack to remove the item from
     * @return StackElement the removed item from the top of the stack
     */
```

- `StackElement peek(Stack *s)`

```
    /**
     * Return the top item from the stack without removing it
     * @param s the stack to retrieve the top item from
     * @return StackElement the item at the top of the stack
     */
```

- `StackElement *createStackElement(char symbol[], int state)`

```
    /**
    @brief Creates a new StackElement with the given symbol and state.
    @param symbol The symbol to be added to the stack element.
    @param state The state to be added to the stack element.
    @return A pointer to the newly created StackElement.
    */
```

- `void printStack(Stack stack, FILE *output_file)`

```
    /**
     * @brief Prints the contents of the stack to the given output file.
     * @param stack The stack to be printed.
     * @param output_file The file to which the stack contents will be
printed.
     * @return void
     */
```

utilsFiles.c :

- `FILE *openFile(char *fileName, char *openMode)`

```
    /**
     * @brief Open a file with a provided name
     *
     * @param fileName The name of the file to open
     * @return FILE* Pointer to the opened file, or NULL if the file does
not exist or no file name was provided
     */
```

- `char *getInputFileName(const char *argv)

```
    /**
     * @brief Creates a new string containing the name of the input file.
     * @param argv The command line argument containing the input file name.
     * @return A pointer to a string containing the name of the input file.
     */
```

- `char *getOutputFileName(char input_file_name[])´

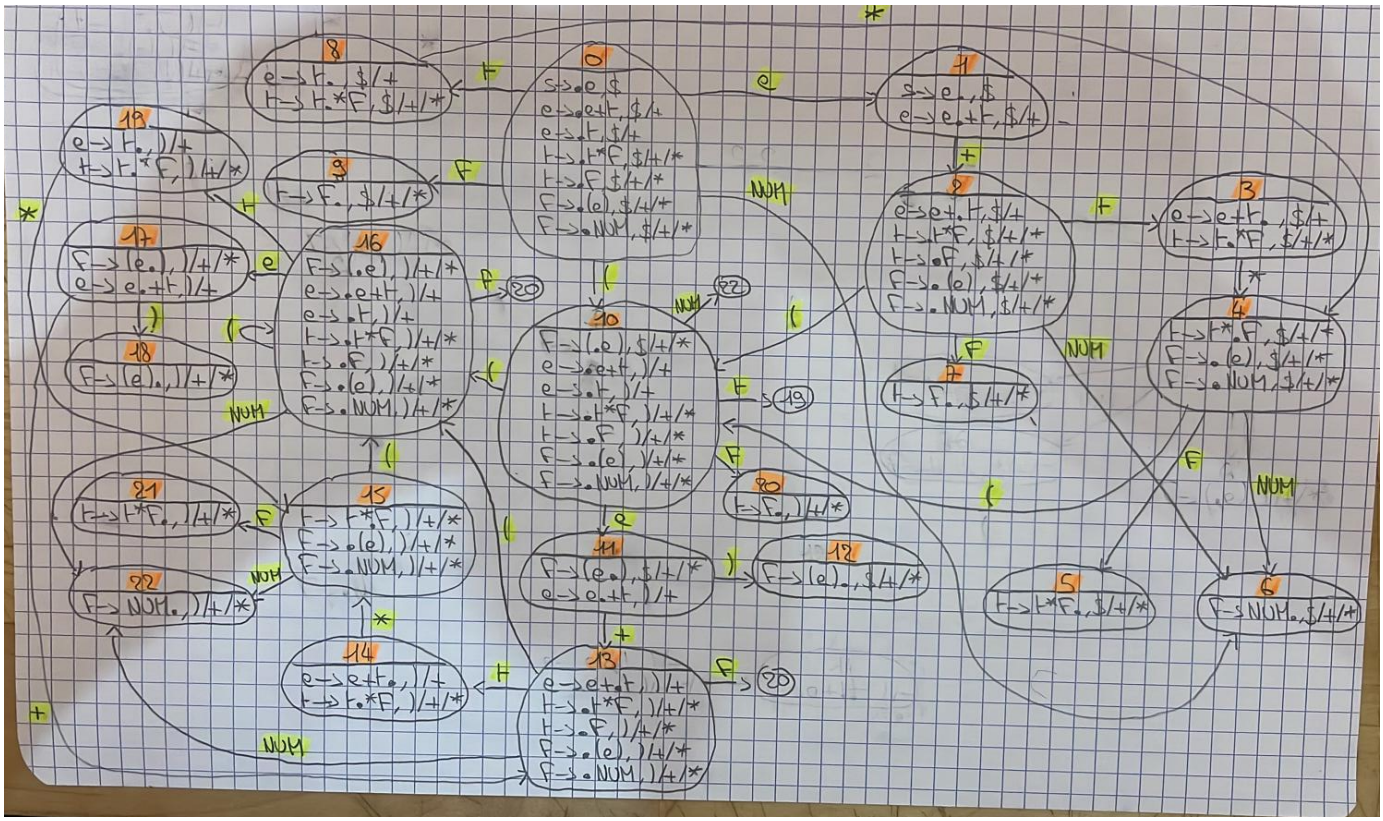
```
    /**
     * Build the name of the output file from the name of the input file by adding scn at the end
     * @param input_file_name an array of character containing the name of the input file
     * @return output_file_name an array of character with the output file name of the form :
     * input_file_name_p3dbg.txt
     */
```

Shift/reduce automaton:

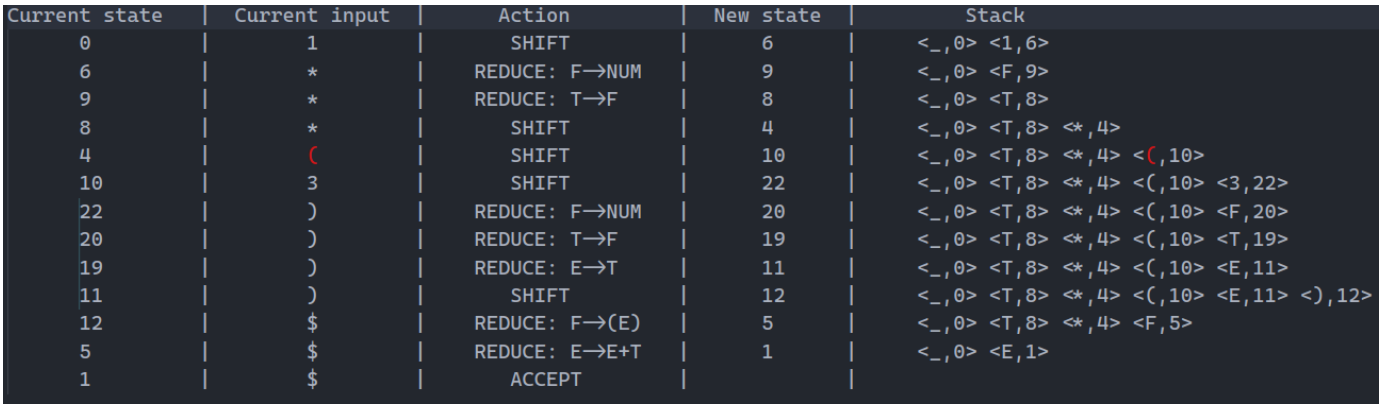
Grammar for the automaton:

1. $s \to e$
2. $e \to e + t$
3. $\mid t$
4. $t \to t * f$
5. $\mid f$
6. $f \to (e)$
7. $\mid NUM$

Automaton:



Output format:



Current state	Current input	Action	New state	Stack
0	1	SHIFT	6	<_,0> <1,6>
6	*	REDUCE: F→NUM	9	<_,0> <F,9>
9	*	REDUCE: T→F	8	<_,0> <T,8>
8	*	SHIFT	4	<_,0> <T,8> <*,4>
4	(	SHIFT	10	<_,0> <T,8> <*,4> <(,10>
10	3	SHIFT	22	<_,0> <T,8> <*,4> <(,10> <3,22>
22	)	REDUCE: F→NUM	20	<_,0> <T,8> <*,4> <(,10> <F,20>
20	)	REDUCE: T→F	19	<_,0> <T,8> <*,4> <(,10> <T,19>
19	)	REDUCE: E→T	11	<_,0> <T,8> <*,4> <(,10> <E,11>
11	)	SHIFT	12	<_,0> <T,8> <*,4> <(,10> <E,11> <),12>
12	$	REDUCE: F→(E)	5	<_,0> <T,8> <*,4> <F,5>
5	$	REDUCE: E→E+T	1	<_,0> <E,1>
1	$	ACCEPT		

This is how the output looks like when we execute the program. The name of the columns is always the same for all the output files regardless of the input files used. The first column means the current state we are in the automaton, the second one is the current lexeme token that we are reading, the third one is the action that is done in every state of the automaton, the fourth one is the new state that we arrive by doing the action and the fifth one is the content of the stack for every state. It is important to know that the action is chosen in function of the current state and the current input.

Action table and goto table:

	action						goto		
state	+	*	(	)	num	$	e	t	f
0			10		6		g1	g8	g9
1	2					accept			
2			10		6			g3	g7
3	re->e+t 0	4				re->e+t 0			
4			10		6				g5
5	rt->t+f 2	rt->t+f 2				rt->t+f 2			
6	rf->num 4/2/0	rf->num 4				rf->num 4/2/0			
7	rt->f 2	rt->f 2				rt->f 2			
8	re->t 0	4				re->t 0			
9	rt->f 0	rt->f 0				rt->f 0			
10			16		22		g11	g19	g20
11	13			12					
12	rf->(e) 0/2/4	rf->(e) 0/2/4				rf->(e) 0/2/4			
13			16		22			g14	g20
14	r e->e+t 10	15		r e->e+t 10					
15			16		22				g21
16			16		22		g17	g19	g20
17	13			18					
18	r f->(e) 10/15/16	r f->(e) 10/15/16		r f->(e) 10/15/16					
19	r e->t 16	15		r e->t 16					
20	r t->f 10	r t->f 10		r t->f 10					
21	r t->t*f 13	r t->t*f 13		r t->t*f 13					
22	r f->num 15/13/16	r f->num 15/13/16		r f->num 15/13/16					

Test inputs:

- `example1.cscn`

```
<1,CAT_NUMBER> <*,CAT_OPERATOR> <(,CAT_ESPECIAL_CHARACTER>
<3,CAT_NUMBER> <),CAT_ESPECIAL_CHARACTER>
```

This Input is accepted, because the sequence of tokens is correct

- Output:

Current state	Current input	Action	New state	Stack
0	1	SHIFT	6	<_,0> <1,6>
6	*	REDUCE: F→NUM	9	<_,0> <F,9>
9	*	REDUCE: T→F	8	<_,0> <T,8>
8	*	SHIFT	4	<_,0> <T,8> <*,4>
4	(	SHIFT	10	<_,0> <T,8> <*,4> <(,10>
10	3	SHIFT	22	<_,0> <T,8> <*,4> <(,10> <3,22>
22	)	REDUCE: F→NUM	20	<_,0> <T,8> <*,4> <(,10> <F,20>
20	)	REDUCE: T→F	19	<_,0> <T,8> <*,4> <(,10> <T,19>
19	)	REDUCE: E→T	11	<_,0> <T,8> <*,4> <(,10> <E,11>
11	)	SHIFT	12	<_,0> <T,8> <*,4> <(,10> <E,11> <),12>
12	$	REDUCE: F→(E)	5	<_,0> <T,8> <*,4> <F,5>
5	$	REDUCE: E→E+T	1	<_,0> <E,1>
1	$	ACCEPT		

- `example2.cscn`

```
<(,CAT_ESPECIAL_CHARACTER> <5,CAT_NUMBER> <*,CAT_OPERATOR> <3,CAT_NUMBER>
<),CAT_ESPECIAL_CHARACTER>
```

This Input is accepted, because the sequence of tokens is correct

- Output:

Current state	Current input	Action	New state	Stack
0	(	SHIFT	10	<_,0> <(,10>
10	5	SHIFT	22	<_,0> <(,10> <5,22>
22	*	REDUCE: F→NUM	20	<_,0> <(,10> <F,20>
20	*	REDUCE: T→F	19	<_,0> <(,10> <T,19>
19	*	SHIFT	15	<_,0> <(,10> <T,19> <*,15>
15	3	SHIFT	22	<_,0> <(,10> <T,19> <*,15> <3,22>
22	)	REDUCE: F→NUM	21	<_,0> <(,10> <T,19> <*,15> <F,21>
21	)	REDUCE: T→T*F	19	<_,0> <(,10> <T,19>
19	)	REDUCE: E→T	11	<_,0> <(,10> <E,11>
11	)	SHIFT	12	<_,0> <(,10> <E,11> <),12>
12	$	REDUCE: F→(E)	9	<_,0> <F,9>
9	$	REDUCE: T→F	8	<_,0> <T,8>
8	$	REDUCE: E→T	1	<_,0> <E,1>
1	$	ACCEPT		

- `example3.cscn`

```
<1,CAT_NUMBER> <-,CAT_OPERATOR> <2,CAT_NUMBER>
```

Input rejected, the sequence of tokens is not correct because of the minus operator