Question: Can you convert this python file into c source code?

Answer: Here is the pure **C (C99 standard)** conversion.

Because C does not have built-in high-level data structures (like maps, sets, or dynamic lists), I have implemented simplified versions using **fixed-size arrays and structs**. This keeps the code self-contained and compile-ready without external libraries.

C Source Code (ll1.c)

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

#define MAX_LINE 1024
#define MAX_SYMBOLS 100
#define MAX_STR_LEN 50
#define MAX_PRODS 200
#define MAX_RHS 20
#define MAX_STACK 1000

// ========================================
// 1. Data Structures
// ========================================

// Simple Set implementation (array of strings)
typedef struct {
    char items[MAX_SYMBOLS][MAX_STR_LEN];
    int count;
} Set;

// Production Rule: LHS -> RHS_1 RHS_2 ...
typedef struct {
    int id;
    char lhs[MAX_STR_LEN];
    char rhs[MAX_RHS][MAX_STR_LEN];
    int rhs_count;
} Production;
```

```c
// Grammar Context
typedef struct {
    Set non_terminals;
    Set terminals;
    char start_symbol[MAX_STR_LEN];
    Production productions[MAX_PRODS];
    int prod_count;
} Grammar;

// Parse Table Entry
typedef struct {
    char nt[MAX_STR_LEN];
    char t[MAX_STR_LEN];
    int prod_id;
} TableEntry;

// LL1 Analyzer Context
typedef struct {
    Grammar *g;
    Set first[MAX_SYMBOLS];  // Map: NonTerminal Index -> Set
    Set follow[MAX_SYMBOLS]; // Map: NonTerminal Index -> Set

    TableEntry table[MAX_PRODS * MAX_SYMBOLS];
    int table_count;
    bool conflict_flag;
} LL1Analyzer;

// Parse Tree Node
typedef struct ParseNode {
    char symbol[MAX_STR_LEN];
    int index;
    struct ParseNode *parent;
    struct ParseNode *children[MAX_RHS];
    int child_count;
} ParseNode;

// ========================================
// 2. Helper Functions
// ========================================

void set_add(Set *s, const char *val) {
    for (int i = 0; i < s->count; i++) {
```

```
        if (strcmp(s->items[i], val) == 0) return;
    }
    strcpy(s->items[s->count++], val);
}

int set_contains(Set *s, const char *val) {
    for (int i = 0; i < s->count; i++) {
        if (strcmp(s->items[i], val) == 0) return 1;
    }
    return 0;
}

// Merge src into dest. Returns 1 if dest changed, 0 otherwise.
int set_update(Set *dest, Set *src) {
    int changed = 0;
    for (int i = 0; i < src->count; i++) {
        if (!set_contains(dest, src->items[i])) {
            set_add(dest, src->items[i]);
            changed = 1;
        }
    }
    return changed;
}

// Remove a specific string from set
void set_discard(Set *s, const char *val) {
    int found_idx = -1;
    for (int i = 0; i < s->count; i++) {
        if (strcmp(s->items[i], val) == 0) {
            found_idx = i;
            break;
        }
    }
    if (found_idx != -1) {
        // Move last item to this spot
        strcpy(s->items[found_idx], s->items[s->count - 1]);
        s->count--;
    }
}

// Get index of a non-terminal in the grammar's NT set
int get_nt_index(Grammar *g, const char *nt) {
    for (int i = 0; i < g->non_terminals.count; i++) {
```

```
        if (strcmp(g->non_terminals.items[i], nt) == 0) return i;
    }
    return -1;
}

// Trim whitespace
char* trim(char* str) {
    char* end;
    while(isspace((unsigned char)*str)) str++;
    if(*str == 0) return str;
    end = str + strlen(str) - 1;
    while(end > str && isspace((unsigned char)*end)) end--;
    *(end+1) = 0;
    return str;
}

// ========================================
// 3. Grammar Loading
// ========================================

void load_grammar(Grammar *g, const char *filename) {
    FILE *f = fopen(filename, "r");
    if (!f) {
        printf("Error: Grammar file '%s' not found.\n", filename);
        exit(1);
    }

    char line[MAX_LINE];
    g->non_terminals.count = 0;
    g->terminals.count = 0;
    g->prod_count = 0;
    int reading_prods = 0;

    while (fgets(line, sizeof(line), f)) {
        char *l = trim(line);
        if (strlen(l) == 0) continue;

        if (strncmp(l, "N =", 3) == 0) {
            char *content = strchr(l, '=') + 1;
            char *token = strtok(content, " \t\n");
            while (token) {
                set_add(&g->non_terminals, token);
                token = strtok(NULL, " \t\n");
```

```
        }
    }
    else if (strncmp(l, "E =", 3) == 0) {
        char *content = strchr(l, '=') + 1;
        char *token = strtok(content, " \t\n");
        while (token) {
            set_add(&g->terminals, token);
            token = strtok(NULL, " \t\n");
        }
    }
    else if (strncmp(l, "S =", 3) == 0) {
        char *content = strchr(l, '=') + 1;
        strcpy(g->start_symbol, trim(content));
    }
    else if (strncmp(l, "P =", 3) == 0) {
        reading_prods = 1;
    }
    else if (reading_prods && strstr(l, "->")) {
        char *arrow = strstr(l, "->");
        *arrow = '\0'; // Split string
        char *lhs = trim(l);
        char *rhs_part = arrow + 2;

        Production *p = &g->productions[g->prod_count];
        p->id = g->prod_count + 1;
        strcpy(p->lhs, lhs);
        p->rhs_count = 0;

        char *token = strtok(rhs_part, " \t\n");
        while (token) {
            strcpy(p->rhs[p->rhs_count++], token);
            token = strtok(NULL, " \t\n");
        }
        g->prod_count++;
    }
  }
  fclose(f);

  // Infer extra terminals from productions if not in N
  for (int i = 0; i < g->prod_count; i++) {
    for (int j = 0; j < g->productions[i].rhs_count; j++) {
        char *sym = g->productions[i].rhs[j];
        if (strcmp(sym, "epsilon") != 0 && !set_contains(&g->non_terminals, sym)) {
```

```
            set_add(&g->terminals, sym);
        }
    }
  }
}

// ========================================
// 4. FIRST & FOLLOW Sets
// ========================================

// Calculate FIRST set for a sequence of symbols
void get_seq_first(LL1Analyzer *ana, char rhs[][MAX_STR_LEN], int count, Set *result) {
  result->count = 0;
  if (count == 0) {
    set_add(result, "epsilon");
    return;
  }

  char *first_sym = rhs[0];
  if (strcmp(first_sym, "epsilon") == 0) {
    set_add(result, "epsilon");
  } else if (set_contains(&ana->g->terminals, first_sym)) {
    set_add(result, first_sym);
  } else {
    // It's a NonTerminal
    int idx = get_nt_index(ana->g, first_sym);
    Set *nt_first = &ana->first[idx];
    set_update(result, nt_first);

    if (set_contains(nt_first, "epsilon")) {
      Set remaining_first;
      // Recursion for rest of sequence
      // Construct a temporary array for the rest
      char sub_rhs[MAX_RHS][MAX_STR_LEN];
      for(int k=0; k < count-1; k++) strcpy(sub_rhs[k], rhs[k+1]);

      get_seq_first(ana, sub_rhs, count - 1, &remaining_first);
      set_update(result, &remaining_first);

      if (!set_contains(&remaining_first, "epsilon")) {
        set_discard(result, "epsilon");
      }
    }
```

```
    }
}

void build_first(LL1Analyzer *ana) {
    // Initialize
    for (int i = 0; i < ana->g->non_terminals.count; i++) {
        ana->first[i].count = 0;
    }

    int changed = 1;
    while (changed) {
        changed = 0;
        for (int i = 0; i < ana->g->prod_count; i++) {
            Production *p = &ana->g->productions[i];
            int lhs_idx = get_nt_index(ana->g, p->lhs);

            Set rhs_first;
            get_seq_first(ana, p->rhs, p->rhs_count, &rhs_first);

            if (set_update(&ana->first[lhs_idx], &rhs_first)) {
                changed = 1;
            }
        }
    }
}

void build_follow(LL1Analyzer *ana) {
    // Initialize
    for (int i = 0; i < ana->g->non_terminals.count; i++) {
        ana->follow[i].count = 0;
    }
    // Add $ to Start Symbol
    int start_idx = get_nt_index(ana->g, ana->g->start_symbol);
    set_add(&ana->follow[start_idx], "$");

    int changed = 1;
    while (changed) {
        changed = 0;
        for (int i = 0; i < ana->g->prod_count; i++) {
            Production *p = &ana->g->productions[i];
            int lhs_idx = get_nt_index(ana->g, p->lhs);

            for (int j = 0; j < p->rhs_count; j++) {
```

```
            char *sym = p->rhs[j];
            if (set_contains(&ana->g->non_terminals, sym)) {
                int sym_idx = get_nt_index(ana->g, sym);

                // Beta is everything after j
                char beta[MAX_RHS][MAX_STR_LEN];
                int beta_len = 0;
                for(int k=j+1; k < p->rhs_count; k++) {
                    strcpy(beta[beta_len++], p->rhs[k]);
                }

                Set beta_first;
                get_seq_first(ana, beta, beta_len, &beta_first);

                // Add beta_first - epsilon
                for(int k=0; k<beta_first.count; k++) {
                    if(strcmp(beta_first.items[k], "epsilon") != 0) {
                        if(!set_contains(&ana->follow[sym_idx], beta_first.items[k])) {
                            set_add(&ana->follow[sym_idx], beta_first.items[k]);
                            changed = 1;
                        }
                    }
                }

                if (beta_len == 0 || set_contains(&beta_first, "epsilon")) {
                    if (set_update(&ana->follow[sym_idx], &ana->follow[lhs_idx])) {
                        changed = 1;
                    }
                }
            }
        }
    }
}

// ========================================
// 5. Parsing Table
// ========================================

void insert_table(LL1Analyzer *ana, const char *nt, const char *t, int prod_id) {
    // Check conflict
    for (int i = 0; i < ana->table_count; i++) {
        if (strcmp(ana->table[i].nt, nt) == 0 && strcmp(ana->table[i].t, t) == 0) {
```

```
            if (ana->table[i].prod_id != prod_id) {
                printf("[CONFLICT] At (%s, %s): Rules %d and %d\n", nt, t, ana->table[i].prod_id,
prod_id);
                ana->conflict_flag = true;
            }
            return;
        }
    }
    // Insert
    strcpy(ana->table[ana->table_count].nt, nt);
    strcpy(ana->table[ana->table_count].t, t);
    ana->table[ana->table_count].prod_id = prod_id;
    ana->table_count++;
}

void build_table(LL1Analyzer *ana) {
    ana->table_count = 0;
    ana->conflict_flag = false;

    for (int i = 0; i < ana->g->prod_count; i++) {
        Production *p = &ana->g->productions[i];
        Set rhs_first;
        get_seq_first(ana, p->rhs, p->rhs_count, &rhs_first);

        for (int k = 0; k < rhs_first.count; k++) {
            char *term = rhs_first.items[k];
            if (strcmp(term, "epsilon") != 0) {
                insert_table(ana, p->lhs, term, p->id);
            }
        }

        if (set_contains(&rhs_first, "epsilon")) {
            int lhs_idx = get_nt_index(ana->g, p->lhs);
            Set *fol = &ana->follow[lhs_idx];
            for (int k = 0; k < fol->count; k++) {
                insert_table(ana, p->lhs, fol->items[k], p->id);
            }
        }
    }
}

int get_table_entry(LL1Analyzer *ana, char *nt, char *t) {
    for (int i = 0; i < ana->table_count; i++) {
```

```
        if (strcmp(ana->table[i].nt, nt) == 0 && strcmp(ana->table[i].t, t) == 0) {
            return ana->table[i].prod_id;
        }
    }
    return -1;
}

// ========================================
// 6. Parser
// ========================================

typedef struct {
    char items[MAX_STACK][MAX_STR_LEN];
    int top;
} Stack;

void push(Stack *s, const char *val) {
    if (s->top < MAX_STACK - 1) strcpy(s->items[++s->top], val);
}
void pop(Stack *s) {
    if (s->top >= 0) s->top--;
}
char* peek(Stack *s) {
    if (s->top >= 0) return s->items[s->top];
    return NULL;
}

// Returns 1 on success, 0 on failure
int parse_sequence(LL1Analyzer *ana, const char *filename, int *output_band, int *out_len) {
    // Load sequence
    FILE *f = fopen(filename, "r");
    if (!f) {
        printf("Error: Sequence file '%s' not found.\n", filename);
        return 0;
    }

    char input_seq[MAX_STACK][MAX_STR_LEN];
    int input_count = 0;
    char buffer[MAX_LINE];
    while (fscanf(f, "%s", buffer) == 1) {
        strcpy(input_seq[input_count++], buffer);
    }
    strcpy(input_seq[input_count++], "$"); // Append end marker
```

```
    fclose(f);

    Stack work_stack;
    work_stack.top = -1;
    push(&work_stack, "$");
    push(&work_stack, ana->g->start_symbol);

    int input_idx = 0;
    *out_len = 0;

    while (1) {
        char *top_work = peek(&work_stack);
        char *top_input = input_seq[input_idx];

        if (strcmp(top_work, "$") == 0 && strcmp(top_input, "$") == 0) {
            return 1;
        }

        if (strcmp(top_work, top_input) == 0) {
            pop(&work_stack);
            input_idx++;
        } else {
            int prod_id = get_table_entry(ana, top_work, top_input);
            if (prod_id != -1) {
                output_band[(*out_len)++] = prod_id;
                pop(&work_stack);

                // Find production by ID (simple search)
                Production *prod = NULL;
                for(int i=0; i<ana->g->prod_count; i++) {
                    if (ana->g->productions[i].id == prod_id) {
                        prod = &ana->g->productions[i];
                        break;
                    }
                }

                // Push RHS reversed (if not epsilon)
                if (prod) {
                    if (!(prod->rhs_count == 1 && strcmp(prod->rhs[0], "epsilon") == 0)) {
                        for (int k = prod->rhs_count - 1; k >= 0; k--) {
                            push(&work_stack, prod->rhs[k]);
                        }
                    }
```

```
            }
        } else {
            printf("Syntax Error at token: '%s'. Expected: ", top_input);
            // Print expected tokens for top_work
            for (int i = 0; i < ana->table_count; i++) {
                if (strcmp(ana->table[i].nt, top_work) == 0) {
                    printf("%s ", ana->table[i].t);
                }
            }
            printf("\n");
            return 0;
        }
    }
  }
}

// ========================================
// 7. Tree Building
// ========================================

ParseNode* create_node(const char *sym) {
  ParseNode *n = (ParseNode*)malloc(sizeof(ParseNode));
  strcpy(n->symbol, sym);
  n->index = 0;
  n->parent = NULL;
  n->child_count = 0;
  return n;
}

// Recursive helper to build tree
ParseNode* recursive_build(Grammar *g, const char *curr_sym, int *prod_ids, int *current_pid_idx, int
total_pids) {
  ParseNode *node = create_node(curr_sym);

  if (set_contains(&g->terminals, curr_sym) || strcmp(curr_sym, "epsilon") == 0) {
      return node;
  }

  if (*current_pid_idx >= total_pids) return node;

  int prod_id = prod_ids[(*current_pid_idx)++];

  // Find production
```

```
    Production *prod = NULL;
    for(int i=0; i<g->prod_count; i++) {
        if(g->productions[i].id == prod_id) {
            prod = &g->productions[i];
            break;
        }
    }

    if (!prod) return node;

    if (prod->rhs_count == 1 && strcmp(prod->rhs[0], "epsilon") == 0) {
        ParseNode *child = create_node("epsilon");
        child->parent = node;
        node->children[node->child_count++] = child;
    } else {
        for (int i = 0; i < prod->rhs_count; i++) {
            ParseNode *child = recursive_build(g, prod->rhs[i], prod_ids, current_pid_idx,
total_pids);
            child->parent = node;
            node->children[node->child_count++] = child;
        }
    }
    return node;
}

void print_tree_bfs(ParseNode *root) {
    if (!root) return;

    // Assign Indexes BFS
    ParseNode* queue[MAX_STACK];
    int front = 0, rear = 0;

    queue[rear++] = root;
    int idx_counter = 1;

    while(front < rear) {
        ParseNode *curr = queue[front++];
        curr->index = idx_counter++;
        for(int i=0; i<curr->child_count; i++) {
            queue[rear++] = curr->children[i];
        }
    }
```

```
    // Print
    printf("\n%-10s %-20s %-10s %-10s\n", "Index", "Symbol", "Father", "Sibling");
    printf("-------------------------------------------------------\n");

    front = 0; rear = 0;
    queue[rear++] = root;

    while(front < rear) {
        ParseNode *curr = queue[front++];

        int father_idx = (curr->parent) ? curr->parent->index : 0;
        int sibling_idx = 0;

        // Find right sibling
        if (curr->parent) {
            for(int i=0; i < curr->parent->child_count; i++) {
                if (curr->parent->children[i] == curr) {
                    if (i + 1 < curr->parent->child_count) {
                        sibling_idx = curr->parent->children[i+1]->index;
                    }
                    break;
                }
            }
        }

        printf("%-10d %-20s %-10d %-10d\n", curr->index, curr->symbol, father_idx, sibling_idx);

        for(int i=0; i<curr->child_count; i++) {
            queue[rear++] = curr->children[i];
        }
    }
}

// ========================================
// 8. Main
// ========================================

int main() {
    // ---------------------------------------------------
    // REQUIREMENT 1: Seminar Grammar
    // ---------------------------------------------------
    printf("\n============================================================\n");
    printf(" REQUIREMENT 1: SEMINAR GRAMMAR & SEQUENCE\n");
```

```
    printf("============================================================\n");

    Grammar g_sem;
    LL1Analyzer ana_sem;
    ana_sem.g = &g_sem;

    load_grammar(&g_sem, "seminar_grammar.txt");
    build_first(&ana_sem);
    build_follow(&ana_sem);
    build_table(&ana_sem);

    if (ana_sem.conflict_flag) {
        printf("[FAIL] Seminar Grammar is not LL(1).\n");
    } else {
        printf("[OK] Seminar Grammar is LL(1). Parsing 'seq.txt'...\n");
        int output_band[MAX_STACK];
        int out_len = 0;
        if (parse_sequence(&ana_sem, "seq.txt", output_band, &out_len)) {
            printf("Sequence parsed successfully!\nOutput (Production String): [");
            for(int i=0; i<out_len; i++) printf("%d%s", output_band[i], (i<out_len-1)?", ":"");
            printf("]\n");
        } else {
            printf("Failed to parse seminar sequence.\n");
        }
    }

    // ------------------------------------------------------
    // REQUIREMENT 2: DSL Grammar
    // ------------------------------------------------------
    printf("\n============================================================\n");
    printf(" REQUIREMENT 2: DSL GRAMMAR & PIF\n");
    printf("============================================================\n");

    Grammar g_dsl;
    LL1Analyzer ana_dsl;
    ana_dsl.g = &g_dsl;

    load_grammar(&g_dsl, "dsl_grammar.txt");
    build_first(&ana_dsl);
    build_follow(&ana_dsl);
    build_table(&ana_dsl);

    if (ana_dsl.conflict_flag) {
```

```
        printf("[FAIL] DSL Grammar is not LL(1).\n");
    } else {
        printf("[OK] DSL Grammar is LL(1). Parsing 'pif.txt'...\n");
        int output_band[MAX_STACK];
        int out_len = 0;
        if (parse_sequence(&ana_dsl, "pif.txt", output_band, &out_len)) {
            printf("PIF parsed successfully!\n");

            // Build Tree
            int pid_idx = 0;
            ParseNode *root = recursive_build(&g_dsl, g_dsl.start_symbol, output_band, &pid_idx,
out_len);
            print_tree_bfs(root);

            // Note: In C we'd typically define a free_tree() function here to clean up mallocs,
            // but we skip it for brevity in this specific assignment scope.
        } else {
            printf("Failed to parse PIF.\n");
        }
    }

    return 0;
}
```