جامعة الأمير سلطان
PRINCE SULTAN
UNIVERSITY

Programming Languages (CS320)

Lexical & Syntax Analysis Project

Lama Alghzzi | 220410092

Norah Alrubayan | 220410543

Sara Alhargan | 219410328

Dr. Siwar Rekik

Table of Contents

Part1: Lexical Analysis

- The lexical analyser recognizes all the tokens correctly
- Reports illegal identifiers or illegal operators

Main.c input

```
x = 5
sum = x * 3 / 5
if sum > 1 then sum = sum + 1 else sum = sum - 1
```

Output:

```
Identifier: x
Operator: =
Integer literal: 5
Identifier: sum
Operator: =
Identifier: x
Operator: *
Integer literal: 3
Operator: /
Integer literal: 5
Keyword: if
Identifier: sum
Operator: >
Integer literal: 1
Identifier: then
Identifier: sum
Operator: =
Identifier: sum
Operator: +
Integer literal: 1
Keyword: else
Identifier: sum
Operator: =
Identifier: sum
Operator: -
Integer literal: 1
```

In this run, all the tokens were recognized correctly. We noticed that it is sensitive to spaces. For example, when we put x=5 without a space, it considers it undefined.

Tokens are divided into:
Identifier, Operator, Integer literal and Keyword.

Main.c input

```
x = 5
2sum = x * 3 / 5
if sum > 1 then sum = sum + 1 else sum = sum - 1
```

Output:

```
Identifier: x
Operator: =
Integer literal: 5
illegal identifier: 2sum
Operator: =
Identifier: x
Operator: *
Integer literal: 3
Operator: /
Integer literal: 5
Keyword: if
Identifier: sum
Operator: >
Integer literal: 1
Identifier: then
Identifier: sum
Operator: =
Identifier: sum
```

In this run, we tried to start the identifier with a number such as *2sum* and it was reported as an illegal identifier.

Main.c input

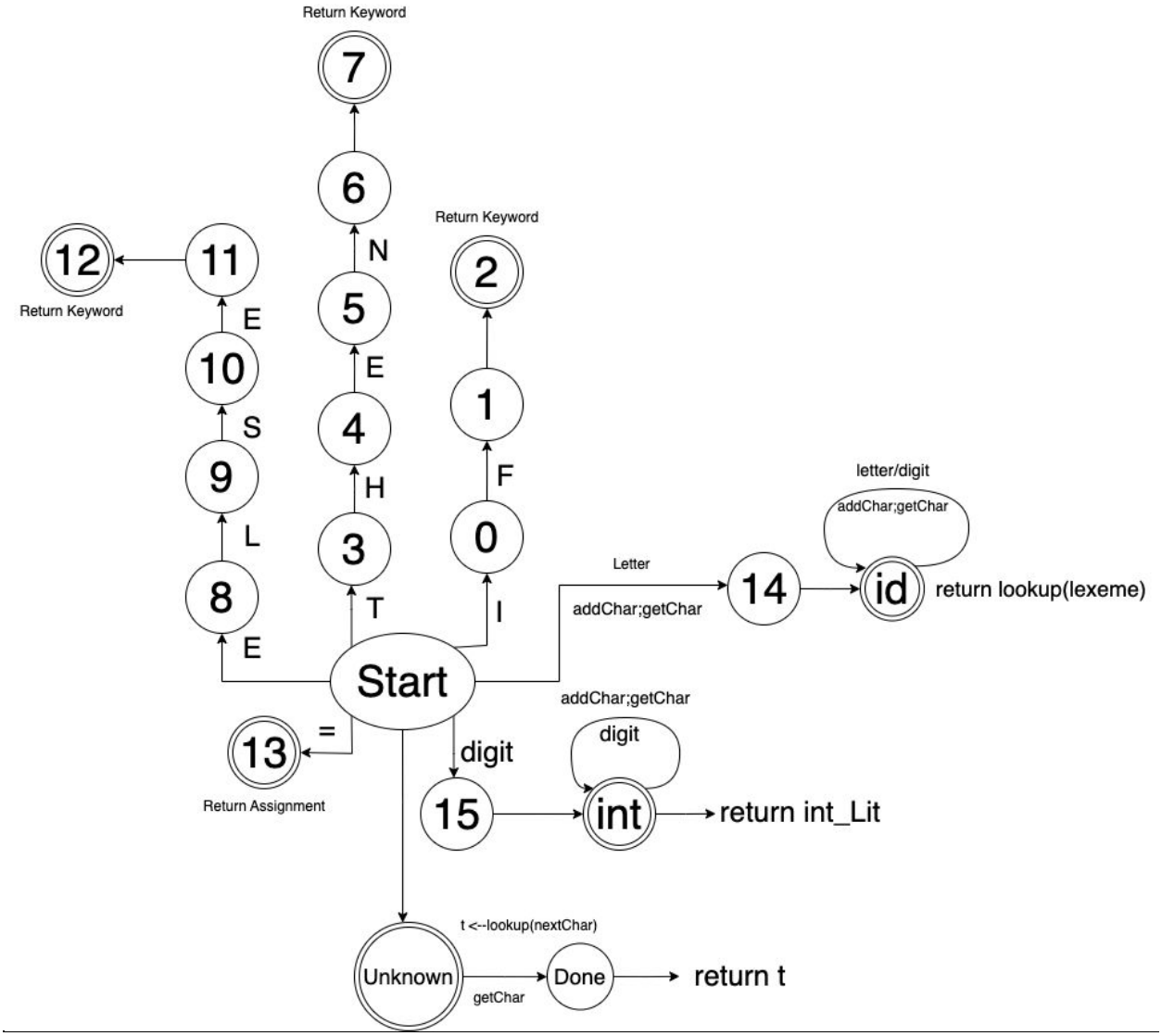
```
x = 5
sum = x ÷ 3 / 5
if sum > 1 then sum = sum + 1 else sum = sum - 1
```

Output:

```
Identifier: x
Operator: =
Integer literal: 5
Identifier: sum
Operator: =
Identifier: x
illegal operator: ÷
Integer literal: 3
Operator: /
Integer literal: 5
Keyword: if
Identifier: sum
Operator: >
Integer literal: 1
Identifier: then
Identifier: sum
Operator: =
Identifier: sum
```

In this run, we used the normal division sign -which is not used in programming syntax- and it was reported as an illegal operator.

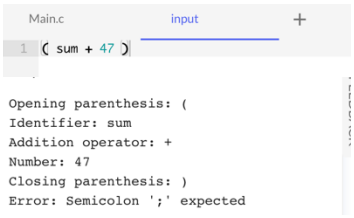
Make appropriate changes to the state transition diagram:



Part2: Syntax Analysis

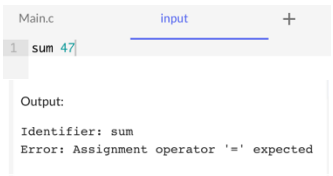
- Missing operator error is detected and reported
- Missing right bracket is detected and reported
- Missing left bracket is detected and reported

To implement error handling in the recursive-descent parser, we can modify each parsing function to call the error function when an unexpected token is encountered.



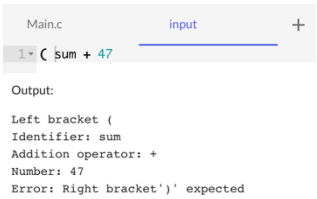
; Missing

Adding the semicolon at the end of the statement is necessary to indicate the end of the statement and avoid any syntax errors.



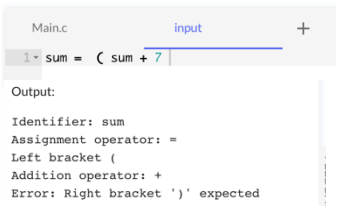
= Missing

The correct syntax to assign a value to the variable "sum" is to include an equals sign followed by the value to be assigned, such as "sum = 0;".



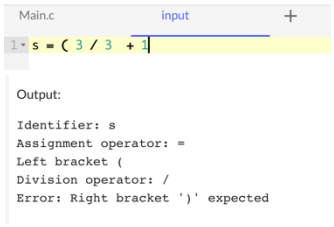
) Missing

The syntax is incomplete and cannot be evaluated by C due to a missing closing parenthesis.



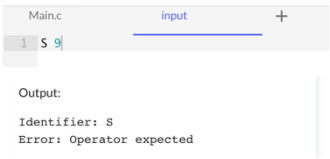
) Missing

The missing ')' in this code snippet is likely due to an unmatched opening bracket '(' on the left side of the addition operator '+'. Adding the closing bracket ')' after the number being added should resolve the error.



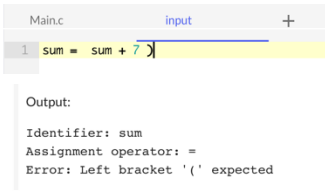
) Missing

The left bracket is opened but not closed, leading to a syntax error due to a missing closing parenthesis.



Operator Missing

The code is trying to perform an operation with the variable "S" without specifying an operator, resulting in the error message "Operator expected".



(Missing

The error message indicates that a left bracket is expected after the assignment operator (=) for the variable "sum". This could be due to a syntax error, incorrect character, or typo.

Appendix

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

#define MAX_TOKEN_LENGTH 100

int is_keyword(char* token) {
    static const char* keywords[] = {"if", "else", "while", "for", "do",
"switch", "case", "default", "break", "continue", "return"};
    static const int num_keywords = sizeof(keywords) / sizeof(char*);
    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(token, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int is_operator(char* token) {
    static const char* operators[] = {"+", "-", "*", "/", "=", "<", ">",
"<=", ">=", "==", "!=", "&&", "||", "!", "&", "|", "^", "~", "<<", ">>"};
    static const int num_operators = sizeof(operators) / sizeof(char*);
    for (int i = 0; i < num_operators; i++) {
        if (strcmp(token, operators[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int is_identifier(char* token) {
    if (!isalpha(token[0]) && token[0] != '_') {
        return 0;
    }
    for (int i = 1; i < strlen(token); i++) {
        if (!isalnum(token[i]) && token[i] != '_') {
            return 0;
        }
    }
    return 1;
}

int is_integer_literal(char* token) {
    if (!isdigit(token[0])) {
        return 0;
    }
    for (int i = 1; i < strlen(token); i++) {
        if (!isdigit(token[i])) {
            return 0;
        }
    }
    return 1;
```

```
}

int main() {
    FILE* fp = fopen("input.txt", "r");
    if (fp == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    char token[MAX_TOKEN_LENGTH];
    while (fscanf(fp, "%s", token) == 1) {
        if (is_keyword(token)) {
            printf("Keyword: %s\n", token);
        } else if (is_operator(token)) {
            printf("Operator: %s\n", token);
        } else if (is_identifier(token)) {
            printf("Identifier: %s\n", token);
        } else if (is_integer_literal(token)) {
            printf("Integer literal: %s\n", token);
        } else {
            printf("Unknown token: %s\n", token);
        }
    }
    fclose(fp);
    return 0;
}
```

```
void error(const char* message);
void ident(const char* token);
void num(const char* token);
void factor(); void term(); void expr(); void assign(); void parse();

FILE* fp; char token[100];

void error(const char* msg) {
printf("Error: %s\n", msg);
exit(1);
}

void ident(const char* token) {
if (!isalpha(token[0])) {
error("Invalid identifier");
 } else {
printf("Identifier: %s\n", token);
}
}
void num(const char* token) {
int i = 0;
while (token[i] != '\0') {
if (!isdigit(token[i])) {
error("Invalid number");
 }
i++;
 }
printf("Number: %s\n", token);
 }
```

```
void factor() { if (strcmp(token, "(") == 0) {
printf("Opening parenthesis: %s\n", token); fscanf(fp, "%s", token); expr();
if (strcmp(token, ")") != 0) {
error("Closing parenthesis ')' expected");
 }
printf("Closing parenthesis: %s\n", token);
}  else if (isalpha(token[0])) {  ident(token);
}  else if (isdigit(token[0])) {  num(token);
}  else {  error("Invalid factor"); }  fscanf(fp, "%s", token); }
 void term() {
factor();  while (strcmp(token, "*") == 0 || strcmp(token, "/") == 0) {
if (strcmp(token, "*") == 0) {  printf("Multiplication operator: %s\n",
token);
} else {  printf("Division operator: %s\n", token);
}  fscanf(fp, "%s", token);  factor();
 }  }  void expr() {
term();  while (strcmp(token, "+") == 0 || strcmp(token, "-") == 0) {
 if (strcmp(token, "+") == 0) {  printf("Addition operator: %s\n", token);  }
else { printf("Subtraction operator: %s\n", token);
} fscanf(fp, "%s", token);
term();
} }

void assign() { ident(token); fscanf(fp, "%s", token); if (strcmp(token, "=")
!= 0) {
error("Assignment operator '=' expected"); } else {
printf("Assignment operator: %s\n", token); }
fscanf(fp, "%s", token);
expr(); }

void parse() { fscanf(fp, "%s", token);
while (!feof(fp)) {
if (isalpha(token[0])) {
assign();
} else if (strcmp(token, "(") == 0) {
factor();
if (strcmp(token, ";") != 0){
error("Semicolon ';' expected");
} else { printf("Semicolon: %s\n", token);
} } else {
error("Invalid input"); }
fscanf(fp, "%s", token); }
} int main(void) {
fp = fopen("input.txt", "r");
if (fp == NULL) {
printf("Failed to open file\n");
return 1; }
parse();
fclose(fp);
return 0;
}
```

```
enum {
    NUM = 256, ID, TRUE, FALSE
};

static int lookahead;

void error(const char* msg) {
    fprintf(stderr, "Error: %s\n", msg);
}

void match(int t) {
    if (lookahead == t) {
        lookahead = lex();
    } else {
        error("Syntax error: unexpected token");
        while (lookahead != t && lookahead != '\n') {
            lookahead = lex();
        }
    }
}

void factor();
void term();
void expr();
void bool();
void join();
void equality();
void relation();
void expression();

void factor() {
    if (lookahead == '(') {
        match('('); expr(); match(')');
    } else if (lookahead == NUM) {
        match(NUM);
    } else if (lookahead == ID) {
        match(ID);
    } else if (lookahead == TRUE) {
        match(TRUE);
    } else if (lookahead == FALSE) {
        match(FALSE);
    } else {
        error("Syntax error: factor expected");
    }
}

void term() {
    factor();
    while (lookahead == '*' || lookahead == '/') {
        match(lookahead);
        factor();
    }
}

void expr() {
    term();
```

```
    while (lookahead == '+' || lookahead == '-') {
        match(lookahead);
        term();
    }
}

void bool() {
    if (lookahead == TRUE) {
        match(TRUE);
    } else if (lookahead == FALSE) {
        match(FALSE);
    } else {
        error("Syntax error: boolean value expected");
    }
}

void join() {
    equality();
    while (lookahead == '&') {
        match('&');
        equality();
    }
}

void equality() {
    relation();
    while (lookahead == '==' || lookahead == '!=') {
        match(lookahead);
        relation();
    }
}
void relation() {
    expression();
    if (lookahead == '<' || lookahead == '>' || lookahead == LE || lookahead
== GE) {
        match(lookahead);
        expression();
    }
}
void expression() {
    expr();
    if (lookahead == '<' || lookahead == '>' || lookahead == LE || lookahead
== GE || lookahead == EQ || lookahead == NE) {
        match(lookahead);
        expr();
    }
}
void parse() {
    lookahead = lex();
    bool();
    match('\n');
}
int main() {
    parse();
    return 0;}
```

References

Sebesta, R. W. (2016). *Concepts of Programming Languages, Global Edition*. Pearson Higher Ed.