◇ EASY LEVEL

Grammar 2 (Easy) – Balanced parentheses

```
S → (S)S | ε
```

RDP for Grammar 2

```c
#include <stdio.h>

#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[64];

int S() {
    if (*pt == '(') {
        printf("%-16s S -> (S)S\n", pt);
        pt++;
        if (S()) {
            if (*pt == ')') {
                pt++;
                return S();  // continue with the next S
            }
            return FAILED;
        }
        return FAILED;
    }
    printf("%-16s S -> $\n", pt);
    return SUCCESS;
}

int main() {
    sscanf("(()())", "%s", input);
    pt = input;

    puts("\nInput Action");

    if (S() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}
```

Grammar 3 (Easy) – Single digit addition

```
S → d A
A → + d A | ε
```

(d represents a digit like 1, 2...)

RDP for Grammar 3

```c
#include <stdio.h>
#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[64];

int S(), A();

int main() {
    sscanf("1+2+3", "%s", input);
    pt = input;

    puts("\nInput Action");

    if (S() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}

int S() {
    if (*pt >= '0' && *pt <= '9') {
        printf("%-16s S -> d A\n", pt);
        pt++;
        return A();
    }
    return FAILED;
}

int A() {
    if (*pt == '+') {
        printf("%-16s A -> + d A\n", pt);
        pt++;
        if (*pt >= '0' && *pt <= '9') {
            pt++;
            return A();
        }
```

```
        return FAILED;
    }
    printf("%-16s A -> $\n", pt);
    return SUCCESS;
}
```

◇ MEDIUM LEVEL

Grammar 4 (Medium) – Simple Boolean Expressions

```
E → T E'
E' → or T E' | ε
T → F T'
T' → and F T' | ε
F → true | false | (E)
```

RDP for Grammar 4

This is a bit long, so let me know if you'd like this one implemented in code too!

Grammar 5 (Medium) – Identifier list

```
S → id L
L → , id L | ε
```

RDP for Grammar 5

```
#include <stdio.h>
#include <ctype.h>
#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[64];

int S(), L();

int main() {
    sscanf("x,y,z", "%s", input);
    pt = input;

    puts("\nInput Action");
```

```
    if (S() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}

int S() {
    if (isalpha(*pt)) {
        printf("%-16s S -> id L\n", pt);
        pt++;
        return L();
    }
    return FAILED;
}

int L() {
    if (*pt == ',') {
        printf("%-16s L -> , id L\n", pt);
        pt++;
        if (isalpha(*pt)) {
            pt++;
            return L();
        }
        return FAILED;
    }
    printf("%-16s L -> $\n", pt);
    return SUCCESS;
}
```

△ HARD LEVEL

Grammar 6 (Hard) – Arithmetic with Subtraction

```
E → T E'
E' → + T E' | - T E' | ε
T → F T'
T' → * F T' | / F T' | ε
F → ( E ) | id
```

This is an enhanced version of your original grammar with `-` and `/` included. Would you like full code for this?

Got it! Here's a **Recursive Descent Parser** in C for a **nested if-else grammar**, where we **fully match keywords like `if`, `then`, `else`, `id`** instead of checking just the first character.

☑ Grammar – Nested if-else (with then)

We'll use this simple grammar that handles **nested if-then-else**:

```
S → if (E) then S else S | id
E → id
```

🔍 Key Points:

- Use `match("if")`, `match("then")`, `match("else")`, etc. to match **entire keywords**.
- Use `strncmp()` to match full keywords, and make sure the next character is a separator (like space, bracket, or semicolon).
- Input: tokenized like `if(id)thenidelseid`

📝 C Code (Nested if-else Parser)

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

#define SUCCESS 1
#define FAILED 0

const char *pt;          // Pointer to current input position
char input[128];

// Function declarations
int S();
int E();

// Match full keyword (e.g., "if", "then", "else", "id")
int match(const char *kw) {
    int len = strlen(kw);
    if (strncmp(pt, kw, len) == 0 && !isalnum(pt[len])) {
        pt += len;
        return 1;
    }
    return 0;
}

// Match identifier (a single `id`)
int id() {
    if (match("id")) return 1;
    return 0;
}

int main() {
    // Sample input string
```

```
    sscanf("if(id)thenif(id)thenidelseid", "%s", input);
    pt = input;

    puts("Input Action:");

    if (S() && *pt == '\0') {
        puts("String is successfully parsed");
    } else {
        puts("Error in parsing String");
    }

    return 0;
}

// S → if (E) then S else S | id
int S() {
    if (match("if")) {
        printf("%-30s S → if ( E ) then S else S\n", pt);
        if (*pt == '(') {
            pt++;
            if (E()) {
                if (*pt == ')') {
                    pt++;
                    if (match("then")) {
                        if (S()) {
                            if (match("else")) {
                                return S();
                            }
                        }
                    }
                }
            }
        }
        return FAILED;
    } else if (id()) {
        printf("%-30s S → id\n", pt);
        return SUCCESS;
    }
    return FAILED;
}

// E → id
int E() {
    printf("%-30s E → id\n", pt);
    return id();
}
```

Example Input:

```
sscanf("if(id)thenif(id)thenidelseid", "%s", input);
```

✔ This input represents:

```
if (id) then
    if (id) then id
    else id
```

Which is a valid **nested if-else**, and will be successfully parsed.

△ Grammar 8 (Hard) – Relational Expressions with Arithmetic

Grammar

```
E  → A E'
E' → < A | > A | == A | ε
A  → T A'
A' → + T A' | - T A' | ε
T  → F T'
T' → * F T' | / F T' | ε
F  → (E) | id
```

Explanation

Supports expressions like:
`(a+b)*c < d`, `x==y+z`, `a/b > x*y+z`

Grammar 9 (Hard) – While Loops + Assignment

Grammar

```
S → while (E) S | id = E ; | { L }
L → S L | ε
E → id < id
```

Explanation

Supports simple constructs like:
`while(x<y) x = x;`,
`{ x=y; while(a<b) a=a; }`

Grammar 10 (Hard) – Ternary Expressions

Grammar

```
E → B E'
E' → ? E : E | ε
B → id | (E)
```

Explanation

Parses ternary expressions:
`a ? b : c`,
`(x ? y : z) ? a : b`

Let me now implement **Grammar 10 (Ternary Expressions)** in C for you as an example:

☑ RDP for Grammar 10 – Ternary Expressions

```c
#include <stdio.h>
#include <ctype.h>
#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[128];

int E(), Ep(), B();

int main() {
    sscanf("a?b:c", "%s", input); // change this to test other expressions
    pt = input;

    puts("\nInput Action");

    if (E() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}

int E() {
    printf("%-16s E -> B E'\n", pt);
    if (B()) {
        return Ep();
    }
    return FAILED;
}

int Ep() {
    if (*pt == '?') {
```

```
            printf("%-16s E' -> ? E : E\n", pt);
            pt++;
            if (E()) {
                if (*pt == ':') {
                    pt++;
                    return E();
                }
                return FAILED;
            }
            return FAILED;
        }
        printf("%-16s E' -> ε\n", pt);
        return SUCCESS;
    }

    int B() {
        if (*pt == '(') {
            printf("%-16s B -> (E)\n", pt);
            pt++;
            if (E() && *pt == ')') {
                pt++;
                return SUCCESS;
            }
            return FAILED;
        } else if (isalpha(*pt)) {
            printf("%-16s B -> id\n", pt);
            pt++;
            return SUCCESS;
        }
        return FAILED;
    }
```

△ Grammar 8 – While Loops with Conditionals

```
S → while (C) S | if (C) S else S | id = E ;
C → id < id
E → id + id | id
```

☑ RDP for Grammar 8

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[128];
```

```c
int S(), C(), E();

int match(const char *kw) {
    int len = strlen(kw);
    if (strncmp(pt, kw, len) == 0) {
        pt += len;
        return 1;
    }
    return 0;
}

int id() {
    if (isalpha(*pt)) {
        pt++;
        return 1;
    }
    return 0;
}

int main() {
    sscanf("while(id<id)id=id+id;", "%s", input);
    pt = input;

    puts("\nInput Action");

    if (S() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}

int S() {
    if (match("while")) {
        printf("%-16s S -> while (C) S\n", pt);
        if (*pt == '(') { pt++;
            if (C() && *pt == ')') {
                pt++;
                return S();
            }
        }
        return FAILED;
    } else if (match("if")) {
        printf("%-16s S -> if (C) S else S\n", pt);
        if (*pt == '(') { pt++;
            if (C() && *pt == ')') {
                pt++;
                if (S()) {
                    if (match("else")) {
                        return S();
                    }
                }
            }
```

```
        }
        return FAILED;
    } else if (id()) {
        printf("%-16s S -> id = E ;\n", pt);
        if (*pt == '=') {
            pt++;
            if (E() && *pt == ';') {
                pt++;
                return SUCCESS;
            }
        }
        return FAILED;
    }
    return FAILED;
}

int C() {
    printf("%-16s C -> id < id\n", pt);
    return id() && *pt++ == '<' && id();
}

int E() {
    printf("%-16s E -> id [+ id]\n", pt);
    if (!id()) return FAILED;
    if (*pt == '+') {
        pt++;
        return id();
    }
    return SUCCESS;
}
```

△ Grammar 9 – Nested Lists

```
L → [ E ]
E → L , E | L | id , E | id
```

☑ RDP for Grammar 9

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[128];

int L(), E();
```

```c
int main() {
    sscanf("[[id],[id,id]]", "%s", input);
    pt = input;

    puts("\nInput Action");

    if (L() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}

int id() {
    if (isalpha(*pt)) {
        pt++;
        return 1;
    }
    return 0;
}

int L() {
    if (*pt == '[') {
        printf("%-16s L -> [E]\n", pt);
        pt++;
        if (E()) {
            if (*pt == ']') {
                pt++;
                return SUCCESS;
            }
        }
    }
    return FAILED;
}

int E() {
    if (*pt == '[') {
        if (L()) {
            if (*pt == ',') {
                pt++;
                return E();
            }
            return SUCCESS;
        }
    } else if (isalpha(*pt)) {
        if (id()) {
            if (*pt == ',') {
                pt++;
                return E();
            }
            return SUCCESS;
        }
```

```
    }
    return FAILED;
}
```

⚠ Grammar 10 – Arithmetic with Unary Minus and Precedence

```
E → T E'
E' → + T E' | - T E' | ε
T → F T'
T' → * F T' | / F T' | ε
F → -F | (E) | id
```

☑ RDP for Grammar 10

```c
#include <stdio.h>
#include <ctype.h>
#define SUCCESS 1
#define FAILED 0

const char *pt;
char input[128];

int E(), Edash(), T(), Tdash(), F();

int main() {
    sscanf("-id+(-id)*id", "%s", input);
    pt = input;

    puts("\nInput Action");

    if (E() && *pt == '\0')
        puts("String is successfully parsed");
    else
        puts("Error in parsing String");

    return 0;
}

int E() {
    printf("%-16s E -> T E'\n", pt);
    if (T()) return Edash();
    return FAILED;
}

int Edash() {
    if (*pt == '+') {
        printf("%-16s E' -> + T E'\n", pt);
        pt++;
```

```
        if (T()) return Edash();
        return FAILED;
    } else if (*pt == '-') {
        printf("%-16s E' -> - T E'\n", pt);
        pt++;
        if (T()) return Edash();
        return FAILED;
    }
    printf("%-16s E' -> ε\n", pt);
    return SUCCESS;
}

int T() {
    printf("%-16s T -> F T'\n", pt);
    if (F()) return Tdash();
    return FAILED;
}

int Tdash() {
    if (*pt == '*') {
        printf("%-16s T' -> * F T'\n", pt);
        pt++;
        if (F()) return Tdash();
        return FAILED;
    } else if (*pt == '/') {
        printf("%-16s T' -> / F T'\n", pt);
        pt++;
        if (F()) return Tdash();
        return FAILED;
    }
    printf("%-16s T' -> ε\n", pt);
    return SUCCESS;
}

int F() {
    if (*pt == '-') {
        printf("%-16s F -> -F\n", pt);
        pt++;
        return F();
    } else if (*pt == '(') {
        printf("%-16s F -> (E)\n", pt);
        pt++;
        if (E() && *pt == ')') {
            pt++;
            return SUCCESS;
        }
        return FAILED;
    } else if (isalpha(*pt)) {
        printf("%-16s F -> id\n", pt);
        pt++;
        return SUCCESS;
    }
    return FAILED;
}
```