Recursive Descent Parser

Alex S.*

1 The `match(...)` function.

The match function takes the form:

```
int match(string s){
    if(current_token == s){
        current_token = next_token();
        return 1;
    }
    return 0;
}
```

2 Using the `match(...)` function.

With the `match(...)` function, translating LL(1) grammar expressions to actual code becomes relatively easy. For example, gramma expression such as:

```
F -> ( F )
F -> ID
```

can be coded such as:

```
void parse_factor(){
    if(match("(")){
        parse_expr();
        match(")");
    }else
        parse_id();
}
```

A more complicated example would be something like this:

*alex@theparticle.com

```
Ftail -> * F Ftail
Ftail -> / F Ftail
```

Which can be translated into something like:

```
void parse_factortail(){
    if(match("*")){
        parse_factor();
        parse_factortail();
    }else if(match("/")){
        parse_factor();
        parse_factortail();
    }
}
```

Then there are statements, and statement lists...

```
StmtList -> Stmt
StmtList -> Stmt StmtList
```

Which can be coded like this:

```
void parse_statementlist(){
    if(there_is_a_next_token()){
        parse_statement();
        parse_statementlist();
    }
}
```

3 Generating Code

Right along with the parsing, we can also output assembly code.

```
Ttail -> + T Ttail
Ttail -> - T Ttail
```

Can be coded (and generate code) such as:

```
void parse_termtail(){
    if(match("+")){
        parse_term();
        parse_termtail();
        printf("pop ax\n");
        printf("pop cx\n");
        printf("add cx,ax\n");
```

```
        printf("push cx\n");
    }else if(match("-")){
        parse_term();
        parse_termtail();
        printf("pop ax\n");
        printf("pop cx\n");
        printf("sub cx,ax\n");
        printf("push cx\n");
    }
}
```

Note that you can output any language (even C!).

3.1 Handling Comparisons

Handling comparisons can be slightly tricky, due to a conditional statement.

```
Comp -> < Comp CompTail
Comp -> > Comp CompTail
```

Which can be coded via something like the following code. Note that the code compares the last two things on the stack, and pushes either a 0 or a 1 depending on the output of the comparison. How would you improve/simplify this code? (hint, you don't really have to do the comparison).

```
void parse_comp_termtail(){
    int l;
    if(match("<")){
        parse_comp_term();
        parse_comp_termtail();
        printf("pop ax\n");
        printf("pop cx\n");
        printf("cmp cx,ax\n");
        printf("mov ax,0\n");
        printf("jge l%d\n",l = label_count++);
        printf("mov ax,1\n");
        printf("l%d:\n",l);
        printf("push ax\n");
    }else if(match(">")){
        parse_comp_term();
        parse_comp_termtail();
        printf("pop ax\n");
        printf("pop cx\n");
        printf("cmp cx,ax\n");
```

```
        printf("mov ax,0\n");
        printf("jge l%d\n",l = label_count++);
        printf("mov ax,1\n");
        printf("l%d:\n",l);
        printf("push ax\n");
    }
}
```

4 Statements

Handling statements is actually simpler than handling arithmetic expressions.

```
Stmt -> read ID
Stmt -> write ID
Stmt -> while Expr Stmt
Stmt -> if Expr Stmt
Stmt -> { StmtList }
...
```

Can be implemented via:

```
void parse_statement(){
    int k,l,m;
    if(match("read")){
        parse_id();
        ...
    }else if(match("write")){
        parse_expr();
        ...
    }else if(match("while")){
        /* top of loop */
        printf("l%d: \n",l = label_count++);
        parse_expr();
        printf("pop ax\n");
        printf("cmp ax,0\n");
        printf("jne l%d\n",m = label_count++);
        printf("jmp l%d\n",k = label_count++);
        printf("l%d:\n",m);
        parse_statement();
        printf("jmp l%d\n",l);
        printf("l%d:\n",k);
    }else if(match("if")){
        parse_expr();
```

```
        printf("pop ax\n");
        printf("cmp ax,0\n");
        printf("jne l%d\n",m = label_count++);
        printf("jmp l%d\n",l = label_count++);
        printf("l%d:\n",m);
        parse_statement();
        if(match("else")){
            printf("jmp l%d \n",k = label_count++);
            printf("l%d: \n",l);
            parse_statement();
            printf("l%d: \n",k);
        }else
            printf("l%d: \n",l);
    ........
```

In the end, one should be able to write a program like this, and have it be translated into assembly:

```
read upperlimit
testnumber = 2
while testnumber < upperlimit {
    testlimit = testnumber - 1
    divisor = 2
    isprime = 1
    while (divisor < testlimit) * isprime {
        testlimit = testnumber / divisor
        isprime = testnumber - testlimit * divisor
        divisor = divisor + 1
    }
    if isprime
        write testnumber
    testnumber = testnumber + 1
}
```

Which waits for the user to enter a number (upper limit), and then proceeds to display all prime numbers upto that upper limit.