Recursive Descent Parser

Below is a recursive descent parser for the following grammar. In the following, "e" is used to represent the empty string.

```
        P -> SP
        P -> e
        S -> id = E
        E -> TE'
        E' -> +TE'
        E' -> e
        T -> FT'
        T' -> *FT'
        T' -> e
        F -> (E)
        F -> id         // I'm cheating here. I want "id" to match
                        // integer number (-6, 0, 47, ...)
                        // or identifier (x, thisIsAlongName, L2, ...)
```

Start of Program &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

```
/*  The first part of the "program" includes "general" routines that would
    be used for any recursive descent parser  */

int lookahead;

error()
{
        printf("Parsing Error\n");
        exit(1);
}


match(t)
        int t;
{
        if( lookahead == t )
                lookahead = getchar();  /* actually you want the
                                           next TOKEN  */
        else
                error();
}


main()
{
        lookahead = getchar();  /* actually you want the next TOKEN  */

        start();
}
```

```
/* **********  Grammar Dependent Routines  *****************  */

start()
{
        P();    /*  the start symbol of the grammar  */
}

P()
{
        if( lookahead != NULL ) {
                S();
                P(); }
        /*  else P -> e   --- do nothing  */
}

S()
{
        match(id);      // match any possible ID.
        match('=');
        E();
}

E()
{
        T();
        EPrime();
}

EPrime()
{
        if( lookahead == '+' ) {
                match('+');
                T();
                EPrime(); }
}

T()
{
        F();
        TPrime();
}
TPrime()
{
        if(lookahead == '*') {
                match('*');
                F();
                TPrime(); }
        /*  else E' -> e   --- do nothing  */
}

F()
{
        if( lookahead == 'id' )
                match(id);
        else
        {
                match('(');
                E();
                match(')');
        }
}
```