Complier_Proj1

功能

- 词法分析和语法分析

  根据`Appendix_A.pdf`完成所需的基本词法和语法分析，为了方便后续的输出和语法树的构建，对于每个非终结符和终结符抽象成树上的节点，它有如下的结构

```
struct Node {
    char *name;
    SymbolType type;
    int line;
    union {
        int         type_int;
        float       type_float;
        char        type_str[40];
    } val;
    struct Node *sons;
    struct Node *next;
};
```

  其中`name`为符号具体是什么，`type`将符号分为几类，`line`为符号位于程序的第几行，`val`为具体的值，`sons`为子节点，`next`为同层下一个节点

```
typedef enum {
    T_INT,
    T_FLOAT,
    T_ID,
    T_TYPE,
    T_RELOP,
    T_TERMINAL,
    T_NTERMINAL,
    T_NULL
} SymbolType;
```

  在词法分析lexical.l中，为符合要求的终结符生成语法树节点，除了`int;float;id;type`特殊考虑，其他的都是`T_NTERMINAL`类型

  对于终结符很容易处理结点，对于非终结符由于子节点数量不定，所以需要用到可变参来传变量，使用`va_list`，`va_start()`，`va_arg()`等来处理，具体可见`syntax.y`下的

`new_node`函数，有了这些语法分析处理就相对容易，就是重复`new_node`的过程，除了对于`Program`需要特殊处理设置一个根节点，其他的全部都是`$$ = new_node(…)`的形式

- 8进制与16进制数

  在`lexical.l`中增加对`hex`和`oct`匹配的正则表达式和报错匹配，同时在`new_node`时对`name`进行特殊赋值以便在`new_node`中特殊处理，直接采用`sscanf`来得到值而不是复杂的进制转换

```
0(0|([1-7][0-7]*)) { // Octal
  ...
}
0[xX](0|([1-9a-fA-F][0-9a-fA-F]*)) { // Hexadecimal
  ...
}
```

- 浮点数

  支持了科学计数法的浮点数表示，只需要在正常的浮点数后加上`([eE][+-]?{u_int})?`

- 注释

  支持两种注释模式和相关的报错

- 报错

  查阅官方文档中的`error recovery`后完成了`syntax`的报错，对于错误发生的位置用关键字`error`占位并规约并且重写`yyerror()`阻止程序中断

编译

在/Code下使用`make parser`即可成功编译出./parser