

编译原理课程设计

学 院（系）： 计算机科学与技术

学 生 姓 名： 祝磊

学 号： 201472003

班 级： 电计 1402

同 组 人： 胡文博（电计 1402）

大连理工大学

Dalian University of Technology

Table of Contents

1 源语言定义	1
2 词法分析	2
2.1 单词的识别模型-有穷自动机 DFA	2
2.2 词法分析器程序的框架	3
2.3 词法分析器程序的工作过程	4
2.4 词法分析器测试	4
3 语法语义分析	5
3.1 语法语义分析器的框架	6
3.2 语法语义分析器的工作过程	6
3.3 递归下降法的实现——节选代码剖析	7
3.3.1 如何从语法定义写出递归下降子程序	7
3.3.2 递归下降子程序如何调用语义动作程序	7
3.3.3 如何维护和使用变量表	8
3.4 语法语义分析程序测试	9
4 抽象机	11
4.1 抽象机的指令系统	11
4.2 抽象机运行测试	12
5 编辑器	12
6 感想	13

1 源语言定义

- (1) $\langle \text{Program} \rangle ::= \text{main}() \{ \langle \text{DeclarationList} \rangle \langle \text{StatementList} \rangle \}$
- (2) $\langle \text{DeclarationList} \rangle ::= \{ \text{int } \langle \text{idList} \rangle \}$
- (3) $\langle \text{StatementList} \rangle ::= \{ \langle \text{Statement} \rangle \}$
- (4) $\langle \text{idList} \rangle ::= \text{ID} \{ , \text{ID} \};$
- (5) $\langle \text{Statement} \rangle ::= \text{if } \langle \text{IfStatement} \rangle \mid \text{while } \langle \text{WhileStatement} \rangle \mid \text{for } \langle \text{ForStatement} \rangle \mid$
 $\text{read } \langle \text{ReadStatement} \rangle \mid \text{print } \langle \text{PrintStatement} \rangle \mid \{ \langle \text{CompoundStatement} \rangle \mid$
 $(\text{ID} \mid \text{NUM} \mid (\mid ;)) \langle \text{Expression} \rangle$
- (6) $\langle \text{IfStatement} \rangle ::= (\langle \text{NoneEmptyExpression} \rangle) \langle \text{Statement} \rangle [\text{else } \langle \text{Statement} \rangle]$
- (7) $\langle \text{WhileStatement} \rangle ::= (\langle \text{NoneEmptyExpression} \rangle) \langle \text{Statement} \rangle$
- (8) $\langle \text{ForStatement} \rangle ::=$
 $(\langle \text{NoneEmptyExpression} \rangle ; \langle \text{NoneEmptyExpression} \rangle ; \langle \text{NoneEmptyExpression} \rangle) \langle \text{Statement} \rangle$
- (9) $\langle \text{ReadStatement} \rangle ::= \text{ID};$
- (10) $\langle \text{PrintStatement} \rangle ::= \langle \text{NoneEmptyExpression} \rangle$
- (11) $\langle \text{CompoundStatement} \rangle ::= \langle \text{StatementList} \rangle \}$
- (12) $\langle \text{Expression} \rangle ::= \langle \text{NoneEmptyExpression} \rangle ; \mid ;$
- (13) $\langle \text{NoneEmptyExpression} \rangle ::= \text{ID} = \langle \text{BoolExpression} \rangle \mid \langle \text{BoolExpression} \rangle$
- (14) $\langle \text{BoolExpression} \rangle ::=$
 $\langle \text{ArithmeticExpression} \rangle [(> \mid < \mid = \mid < = \mid = = \mid ! =) \langle \text{ArithmeticExpression} \rangle]$
- (15) $\langle \text{ArithmeticExpression} \rangle ::= \langle \text{Term} \rangle \{ (+ \mid -) \langle \text{Term} \rangle \}$
- (16) $\langle \text{Term} \rangle ::= \langle \text{Factor} \rangle \{ (/ \mid *) \langle \text{Factor} \rangle \}$
- (17) $\langle \text{Factor} \rangle ::= (\langle \text{NoneEmptyExpression} \rangle) \mid \text{ID} \mid \text{NUM}$

上述语言定义中，属于终结符的 $\{ , \} , (,)$ 已用红色标出。

需要指出，上述文法中的左递归均已消除，但有一处隐含的左公共因子，在第（13）式中，因而不是 LL(1) 文法。容易证明，该文法是 LL(2) 文法，且只有选择 $\langle \text{NoneEmptyExpression} \rangle$ 的推导式时需要向前看两个符号，可能产生一步回溯。考虑到消去后将产生含义不甚明确的非终结符，我们没有消去该处的左公共因子，而选择了忍受在递归下降中可能产生的回溯。已经指出，只有在解析 $\langle \text{NoneEmptyExpression} \rangle$ 时这种情况才可能发生且最多回退一步，所以是可以接受的。

2 词法分析

在本次课程设计中我和队友各写了一个词法分析器，这里是介绍的是我的词法分析方案。对应的是 NaiveCCompiler/src/compiler/lexicalAnalyzer-Leif 目录下的词法分析器。

2.1 单词的识别模型-有穷自动机 DFA

合法的单词定义如下：

$\langle \text{ID or Keyword} \rangle ::= \langle \text{alphabet} \rangle \{ \langle \text{alphabet} \rangle \mid \langle \text{digits} \rangle \}$

$\langle \text{numbers} \rangle ::= \langle \text{digits} \rangle \{ \langle \text{digits} \rangle \}$

$\langle \text{delimiters}_1 \rangle ::= + \mid - \mid * \mid (\mid) \mid \{ \mid \} \mid ; \mid , \mid :$

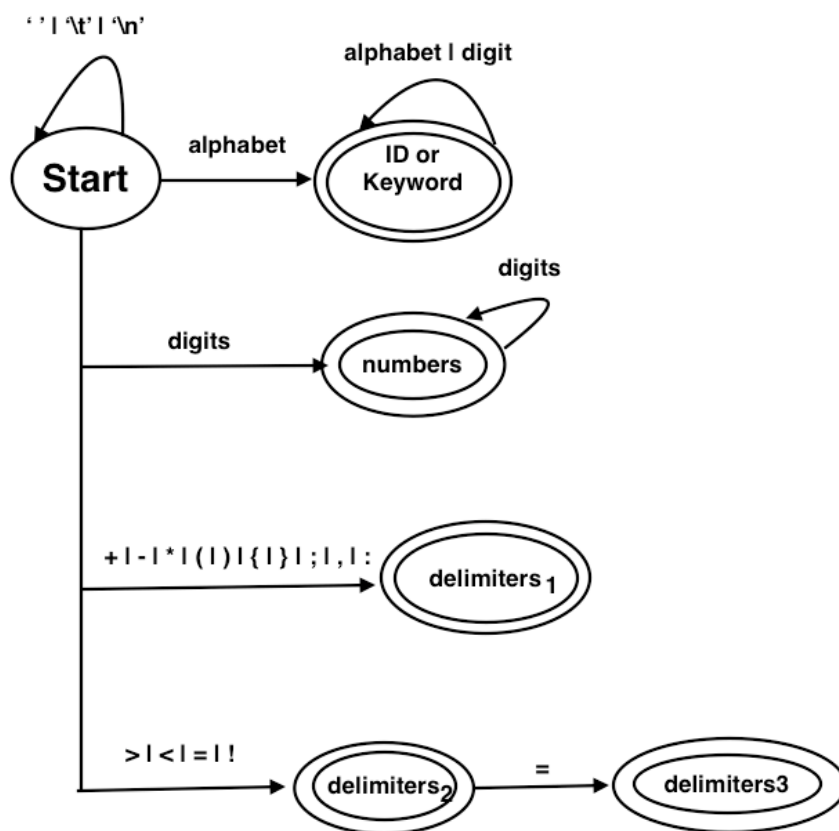
$\langle \text{delimiters}_2 \rangle ::= > \mid < \mid = \mid !$

$\langle \text{delimiters}_3 \rangle ::= >= \mid <= \mid == \mid !=$

$\langle \text{digits} \rangle ::= 0-9$

$\langle \text{alphabet} \rangle ::= a-z \mid A-Z$

识别单词的 DFA



2.2 词法分析器程序的框架

```
#ifndef _LEXICALANLYZER_H_
#define _LEXICALANLYZER_H_
#include <string>
#include <fstream>
#include <vector>
using std::ifstream;
using std::string;
using std::streamoff;
using std::vector;

typedef struct
{
    string type;
    string value;
}Token;

class lexicalAnalyzer
{
    vector<string> keyword; //keyword list
    string delimiters; //delimiters
    ifstream fin; // input file
    unsigned int lineTag;
    unsigned int oldLineTag; //for rollbacking one token
    streamoff oldOffset; //for rollbacking one token
    char ch;
public:
    lexicalAnalyzer(string inputFileName);
    ~lexicalAnalyzer();
    Token next(); //get next token
    void back(); // rollback one token
    unsigned int getLineTag(); // get current line tag
    void analyze(); //the entrance of lexical analysis
};
#endif
```

我将词法分析器封装成了类 lexicalAnalyzer 。

类成员变量包括:

- (1) 一个可以在初始化时定制的关键字表 keyword。
- (2) 一个可以在初始化时定制的分隔符表 delimiters。
- (3) 输入文件流 fin。
- (4) 一个记录行号的无符号整型变量 lineTag。
- (5) 一个行号备份 oldLineTag。前面说过，我们的文法不是 LL(1)文法，可能需要回溯，设置这个 oldLineTag 是为了方便回溯时恢复行号。
- (6) 一个文件流偏移量 oldOffset。设置理由同上。
- (7) 一个用于从输入文件接收字符的 char 类型变量 ch。

类成员函数包括：

- (1) 构造函数和析构函数。
- (2) 返回 Token 的函数 `next()`。每调用一次 `next()`,从文件中读取一个单词的类型和值。这里对自定义的 Token 结构做如下说明：
 - 对于标志符, `type` 字段置为 ID, `value` 字段置为标志符名
 - 对于数字, `type` 字段置为 NUM, `value` 字段置为数字串
 - 对于分隔符和关键字, `type` 字段和 `value` 字段相同, 均为分隔符本身
 - 区分 ID 和关键字的方法是查找、比对关键字表
- (3) 回退函数 `back()`,调用后可以回到上一个 token 读取后的状态。
- (4) `getLineTag()`是 LineTag 的接口, 用于在对象外部获取行号。可供语法分析器调用, 报错时提供出错行号。
- (5) `analyze()`是词法分析的入口, 调用后自动不断调用 `next()`直至到达输入文件结束, 每次获得一个 token 都会将结果打印到屏幕上。

2.3 词法分析器程序的工作过程

- (1) 创建一个词法分析器对象, 构造函数将完成初始化工作
- (2)如果是被语法分析器调用,则是每次由语法分析器调用`next()`函数获取一个 token 进行进一步分析处理。也可以单独使用, 调用 `analyze()` 函数启动自动打印 token 流, 直至文件结束。
- (3) 遇到无法识别的单词或不符合定义的单词时, 词法分析器会抛出异常。被捕获后打印错误信息并提示行号。

2.4 词法分析器测试

词法分析器的实现并不困难, 实现起来不过百余行, 直接阅读源码没有困难。这里不再赘述源码, 直接给出测试结果。

输入文件内容	输出结果
--------	------

<pre>main() { int n; n = 5; while(n>0) { n--; print n; } }</pre>	<pre>[MBA:release & test file Leif\$./lexer correct.nc main main (()) { { int int ID n ; ; ID n = = NUM 5 ; ; while while ((ID n > > NUM 0)) { { ID n - - - - ; ; print print ID n ; ; } } })</pre>
<pre>main() { int n; n = 5x; while(n>0) { n--; print n; } }</pre>	<pre>[MBA:release & test file Leif\$./lexer wrong.nc main main (()) { { int int ID n ; ; ID n = = error: line4: illegal identifier!</pre>

3 语法规义分析

我们采用了递归子程序法进行语法制导的语义分析。

递归子程序法，又称递归下降分析法。思想是：非终极符的文法规则可看成是识别该非终极符的过程定义，针对文法的每一非终极符，根据相应产生式各候选式的结构，为其编写一个递归函数，用来识别该非终极符所表示的语法范畴。

递归下降分析法是一种确定的自顶向下的语法分析方法，其分析过程是从文法开始符号出发，执行一组递归过程，不断向下推导，直到推出句子。使用递归下降法的好处是程序易编写易读，代价是相比于自底向上分析，向下递归时频繁地调用函数，将产生额外的开支，且分析大型的程序时递归栈太深，编译程序容易溢出。

在语法定义的产生式中加入语义动作，构成属性文法。对应地，我们在语法分析程序中插入执行语义动作的函数，就得到了语法制导的语义分析程序，实现起来十分便捷。

下面分析我们编写的语法语义分析程序。

3.1 语法语义分析器的框架

我们的语法分析程序使用 C++ 语言、面向对象方式编写，语法语义分析器被封装成一个类(class)SemanticAnalyzer。

类成员变量包括：

- (1) 一个输入流对象 `fin`，处理待分析的输入文本；
- (2) 一个输出流对象 `fout`，输出目标代码；
- (3) 一个 `string` 对象 `codeOut`，记录输出文件名；
- (4) 一个符号表 `varTable`，记录变量的名字和分配的地址；
- (5) 一个词法分析器对象 `lexer`，从输入的文本中提取单词流；
- (6) 一个 `Token` 对象 `token`，从词法分析器接收单词；
- (7) 一个 `int` 对象 `labelTag`，用于设置标签
- (8) 一个 `int` 对象 `dataAddress`，用于分配变量地址

类成员函数包括三部分：

- (1) 类构造 / 析构函数和语法语义分析入口函数
- (2) 语义动作函数
- (3) 语法单元函数（对应语义分析的各终结符）

参看 `SemanticAnalyzer.h` 头文件可以对我们的语法语义分析器的框架一目了然。

3.2 语法语义分析器的工作过程

- (1) 创建一个语法语义分析器对象，构造函数完成初始化工作
- (2) 调用语法语义分析的入口函数 `analyze()` 启动分析过程
- (3) 从语法单元(终结符)<program>开始递归下降分析整个程序
 - 递归下降时，每调用一次词法分析器，取出一个单词进行匹配，所以我们的编译程序是单次扫描的。
 - 在递归下降中，各语法单元函数（对应语法定义中的非终结符）中会同时调用语法动作函数，实现语法制导的语义分析。
 - 当语法分析遇到错误时，会抛出异常，终止语法语义分析，异常被捕获后向屏幕打印错误信息。

3.3 递归下降法的实现——节选代码剖析

这里我们主要分析下面的问题：

- (1) 如何从语法定义写出递归下降子程序
- (2) 递归下降子程序如何调用语义动作程序
- (3) 变量表如何维护和使用

3.3.1 如何从语法定义写出递归下降子程序

我们先回顾一下 `while` 语句的形式化定义然后看看如何写出语法分析的递归子程序。

语法定义第(7)条：

$$\langle \text{WhileStatement} \rangle ::= (\langle \text{NoneEmptyExpression} \rangle) \langle \text{Statement} \rangle$$

这里没有 `while` 在前是因为 `while` 在第(5)条定义 $\langle \text{Statement} \rangle$ 中已经被匹配过。现在看看它对应的递归子程序：

```
void SemanticAnalyzer::whileStat()
{
    if(!match(_PARENTHESSE_L)) throw 5; //如果不是左括号，抛出错误
    get(token); //进入子程序前要更新一下 token
    noneEmptyExpression(); //分析循环条件，一个非空表达式
    if(!match(_PARENTHESSE_R)) throw 6; //如果不是右括号，抛出错误
    get(token); //进入子程序前要更新一下 token
    statement(); //分析循环体，一个语句（语句包括复合语句）
}
```

3.3.2 递归下降子程序如何调用语义动作程序

上面的子程序仅仅实现了语法分析，要实现语义分析，还要插入一些语义动作。在这之前我们回顾一下 `while` 语句的语义。

`while(循环条件) 循环体`

`while` 语句有两个语句块，一个是循环条件，另一个是循环体。到达 `while` 语句时先计算循环条件是否为真：如果为假，前往出口；如果为真，进入循环体，并在完成循环体后重新转到循环条件。

由于编译器是顺序输出指令，要实现跳转，就需要设置标志(label)，来确定跳转的目标位置（类似 C 语言中的 `goto` 语句）。

加入语义动作的处理 while 语句的子程序如下：

```
void SemanticAnalyzer::whileStat()
{
    if(!match(_PARENTHESE_L)) throw 5;
    get(token);
    int label1 = labelTag++;
    setLabel(label1); //设置 label1 记下循环条件开始的位置
    noneEmptyExpression(); //计算循环条件
    if(!match(_PARENTHESE_R)) throw 6;
    int label2 = labelTag++;
    jz(label2); //如果循环条件为假就直接转向 label2
    get(token);
    statement(); //否则就会顺序执行，进入循环体
    jmp(label1); //循环体结束，跳回循环条件所在位置
    setLabel(label2); //设置 label2,标志出口位置
}
```

这样就实现了语法制导的语义翻译。语义动作函数的行为再此不赘述。参看源码文件 SemanticAnalyzer.cpp。

3.3.3 如何维护和使用变量表

与变量表有关的操作有两个:插入新变量和查找变量地址。在我们的语法语义分析程序中分别对应函数 varDef(string &name)和 lookup(string &name,int &address) 。下面是代码和注释:

```
void SemanticAnalyzer::varDef(string &name)//插入新变量并为之分配地址
{
    for(unsigned long i = 0 ;i < varTable.size(); i++)
    {
        if(varTable[i].name == name) //如果在变量表中发现相同的名字
        {
            throw 12; //抛出错误 12 表示变量重复定义
        }
    }
    varTable.push_back(Record(name,dataAddress));
}
```

```
//没有发现重复定义，就将新变量插入符号表
dataAddress++; //数据地址自增，指向下一空闲位置
}

void SemanticAnalyzer::lookup(string &name,int &address)
{
    for(int i = 0; i < varTable.size(); i++)
    {
        if(name == varTable[i].name) //查找变量名是否在表中
        {
            address = varTable[i].address; //若找到，则返回地址结束
            return;
        }
    }
    throw 13; //找不到时才会进入这一步，抛出错误 13 表示使用了未声明的变量
}

varDef 函数在定义新变量时会被调用(参看源码 idList()子程序),lookup 函数在使用变量时会被调用(参看源码 noneEmptyExpression(),factor(),readStat()三个子程序)。
```

3.4 语法规义分析程序测试

我们首先用 vim 创建了一个用 naïve C 语言编写的测试程序 test.nc, 然后用编译器 ncc 编译, 输出 test.s 伪汇编文件, test.s 可以被后面的抽象机执行。

首先看一下 test.nc 文件的内容。

```
main()
{
    int i,n,j;
    read n;
    if(n < 10)
    {
        j = 1;
        for(i = 1; i <= n; i = i + 1)
        {
            j = j * i;
        }
        print j;
    }
    else
    {
        i = 1;
        j = 1;
        print i;
        print j;
        n = i + j;
        while(n < 100)
        {
            print n;
            i = j;
            j = n;
            n = i + j;
        }
    }
}
```

这段程序实现的功能是读入一个 n ，当 $n < 10$ 时，计算 $n!$ 并输出；当 $n \geq 10$ 时，输出 100 以内的斐波那契数列。程序中包含了顺序，分支，循环三种基本结构。

下面用编译器 ncc compiler 编译：

```
MBA:Debug Leif$ ./ncc test.nc -o test.s
build succeeded!
```

编译器提示编译成功。

现在用 vim 打开并查看编译器输出的伪汇编文件 test.s。输出文件比较长，我们只看看前几行，示意输出文件的形式。

```
IN
STO 1
POP
LOAD 1
IMM 10
LES
JZ 0
IMM 1
STO 2
POP
IMM 1
STO 0
POP
LABEL1:
LOAD 0
LOAD 1
LE
JZ 2
IMM 2
```

直接读这个伪汇程序比较费劲。可以看第四部分抽象机的执行结果来确定编译结果的正确性。

4 抽象机

该部分由我的队友胡文博完成。下面只简要介绍该抽象机的指令系统，因为抽象机机的指令决定了我的语义分析器输出怎样的代码。

4.1 抽象机的指令系统

- **LOAD** **addr** --将地址 **addr** 内存单元中的数据入栈
- **IMM** **num** --将立即数 **num** 入栈
- **STO** **addr** --将栈顶数据存入地址 **addr** 所对应的内存单元
- **POP** --栈顶数据出栈
- **ADD** --将栈顶和次栈顶数据出栈，并将其和入栈
- **SUB** --将栈顶和次栈顶数据出栈，并将次栈顶和栈顶数据的差入栈
- **MULT** --将栈顶和次栈顶数据出栈，并将其积入栈
- **DIV** --将栈顶和次栈顶数据出栈，并将次栈顶和栈顶数据的商入栈
- **JMP** **label** --将 PC 无条件跳转到 **label** 所对应的代码段地址
- **JZ** **label** --首先将栈顶数据出栈，并判定此数据的真假，为真就将 PC 跳转到 **label** 所对应的代码段地址，否则不跳
- **EQ** --将栈顶和次栈顶数据出栈，若次栈顶和栈顶数据相等则将真值入栈，否则将假值入栈
- **NOTEQ** --将栈顶和次栈顶数据出栈，若次栈顶和栈顶数据不等则将真值入栈，否则将假值入栈
- **GT** --将栈顶和次栈顶数据出栈，若次栈顶数据大于栈顶数据则将真值入栈，否则将假值入栈
- **LES** --将栈顶和次栈顶数据出栈，若次栈顶数据小于栈顶数据则将真值入栈，否则将假值入栈
- **GE** --将栈顶和次栈顶数据出栈，若次栈顶数据大于或等于栈顶数据则将真值入栈，否则将假值入栈
- **LE** --将栈顶和次栈顶数据出栈，若次栈顶数据小于或等于栈顶数据则将真值入栈，否则将假值入栈
- **AND** --将栈顶和次栈顶数据出栈，并将其相与的结果入栈
- **OR** --将栈顶和次栈顶数据出栈，并将其相或的结果入栈
- **NOT** --将栈顶数据出栈，若该数据为真则将假值入栈，否则将真值入

栈

- IN --输入指令,用于从标准输入设备获取一个数据并将该数据入栈
- OUT --输出指令,用于向标准输出设备输出栈顶数据并将该数据出栈

4.2 抽象机运行测试

在语法语义分析程序中,我们生成了伪汇编代码文件 `test.s`。这里我们用抽象机执行该文件。结果如下:

```
MBA:Debug Leif$ ./nvm test.s
Please input an integer:
5
Program output: 120
MBA:Debug Leif$ ./nvm test.s
Please input an integer:
11
Program output: 1
Program output: 1
Program output: 2
Program output: 3
Program output: 5
Program output: 8
Program output: 13
Program output: 21
Program output: 34
Program output: 55
Program output: 89
MBA:Debug Leif$
```

上述测试中,我运行了两次 `test.s`,第一次输入 5,输出了 120,即 $5!$;第二次输入了 11,输出了 100 以内的斐波那契数列。输出结果均符合预期。至此可以证明:

- (1) 我们的编译程序将源文件 `test.nc` 正确转换成了伪汇编文件 `test.s`
- (2) 我们的虚拟机文件可以正确执行由编译程序生成的伪汇编文件

5 编辑器

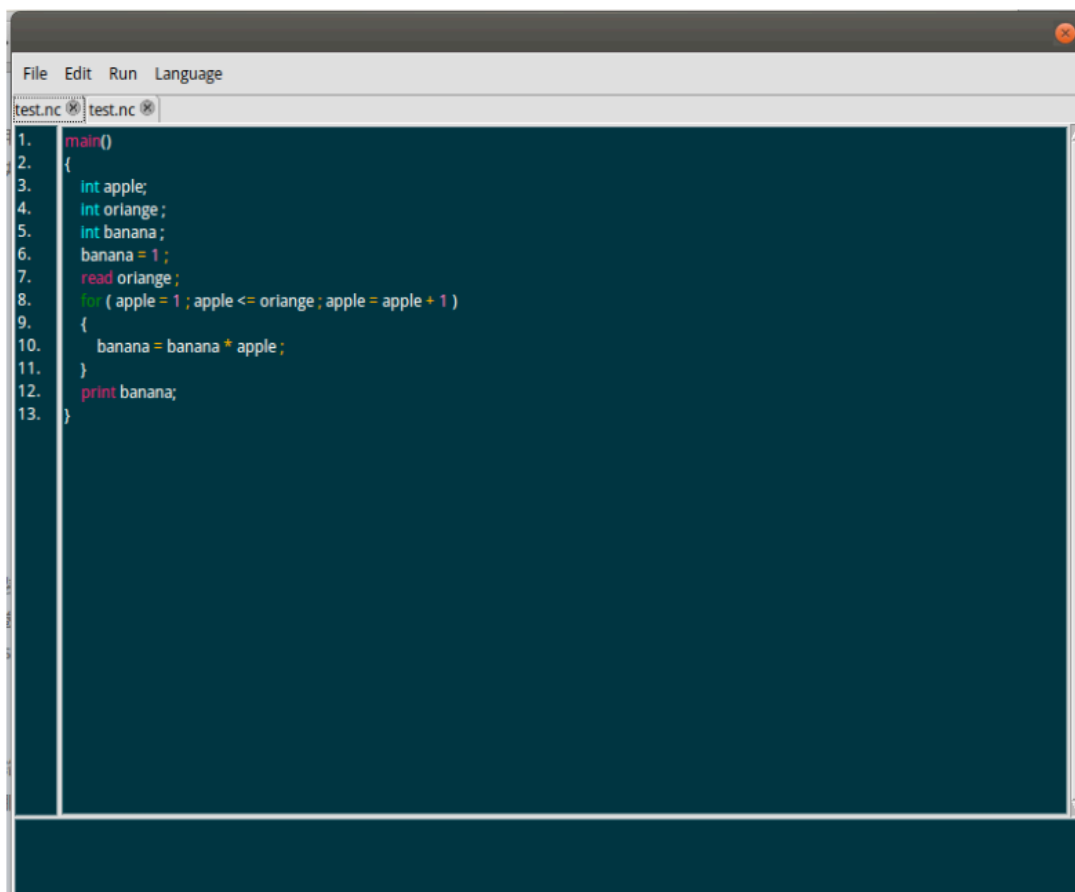
该部分由我的队友胡文博完成。下面只作简要介绍。

(1) 编辑器实现的功能:

- 语法高亮。对函数名、数据类型、关键字、运算符、常数分别用红、蓝、绿、橙、紫五种颜色高亮显示
- 自动补全。基于字典树实现。

- 对接编译器与虚拟机。可在编辑完成后在编辑器界面一键编译，并调出任务台运行。

(2) 界面展示



```
1. main()
2. {
3.     int apple;
4.     int orange;
5.     int banana;
6.     banana = 1;
7.     read orange;
8.     for ( apple = 1 ; apple <= orange ; apple = apple + 1 )
9.     {
10.        banana = banana * apple;
11.    }
12.    print banana;
13. }
```

6 感想

在这次编译原理课程设计中我与队友胡文博一起完成了语言定义开始一个简单的编译器。项目用面向对象的方式组织代码使得我们的代码结构清晰，扩展容易，源码易于阅读。在代码编写过程中，我们使用了 git 进行版本控制和管理，大大方便了合作开发。使用 git 的另一个好处是，我们可以将代码托管 github 上，历史提交记录一目了然。该项目的地址是 <https://github.com/crisb-DUT/NaiveCCompiler>。

由于时间仓促，我们的项目仍有一些不成熟的地方：

- (1) 我们定义的 naïve C 语言相对比较简单，支持的数据类型有限，不支持函数调用；

(2) 初期任务分配不尽合理，词法分析器和语法语义分析器分别由两个人编写，导致在对接时增加了沟通成本。

尽管如此，我仍然对目前的成果还是相对满意。因为写出一个简单的编译器，做出更复杂的版本就只是时间问题。在课程设计中温习了编译原理的基本理论知识，提高了编程能力，在实践中加深了对程序、对编译器的理解。

最后，对本次课程设计过程中老师的耐心指导和队友胡文博的通力合作表示感谢。