小型SysY语言编译器的实现

石依凡 2020202264 2022年6月

目录

1 实验概述

本实验完成了一个从C语言的子集语言（SysY语言）编译生成x86_64 AT&T格式汇编的小型编译器，分3个阶段完成：词法分析、语法分析、语义分析阶段。

词法分析阶段，词法分析器基于LEX分析器，完成从输入SysY语言源代码到机器可识别%token的转化，包括关键字（**K**eyword）、界符（**D**elimeter）、标识符（**I**dentifier）、运算符（**O**perator）、常量字面值（**C**onstant）、空白值（**B**lank）和未知值（**U**nknown）。对于每一个%token，机器建立相应的节点，并将其传给语法分析器。

语法分析阶段，语法分析器完成从输入%token序列，到输出语法树的过程转化，基于YACC，使用自下而上规约的LR方法。语法树节点分为终结符叶子节点，即词法分析器传来的%token节点，和规约形成的非终结符%node节点。

语义分析阶段，语义分析器基于语法分析器，在规约的过程中完成分配寄存器、回填真假链、传送节点值、产生汇编码等语义动作，最终输出为完整的AT&T格式x86_64汇编码源文件。

2 词法分析部分

本部分需要利用LEX语言，产生能够解析SysY语言单词的二进制可执行文件，并将其一一分类为指定的若干类别，然后在接下来的部分中继续拓展使用。本部分巩固了所学的正则表达式、有穷自动机和文法三种等价写法的相关联性，使用了现代工具实践词法分析，并生成了能够在语法分析、语义分析乃至编译器编写中进一步使用的可执行文件。

2.1 流程说明

本部分利用LEX语言和flex工具，在类Unix平台（MacOS）上进行编程、编译、执行。主要分为以下几个部分：

- 学习LEX语言。熟悉LEX语言的写法，阅读相关材料。
- 编写LEX语言脚本。利用LEX语言对正则表达式所做相关语法规定，编写正则表达式来识别相关单词，并编写C语言处理过程块。
- 解释、翻译LEX语言脚本。使用flex 2.6.4 Apple (flex-34)工具将LEX脚本翻译成为C语言源代码。

- 编译C语言代码，形成可执行程序。使用C语言编译器Apple clang version 13.0.0 (clang-1300.0.29.30)将flex工具生成的C语言代码编译生成二进制可执行文件，即需要使用的词法分析器。

- 对事先准备的SYSY语言测试脚本使用编译产生的词法分析器进行词法分析测试，调试程序，重复上述过程。

- 总结、报告。

2.2 LEX脚本设计

LEX脚本分为四个部分：预处理、常量定义、行为定义和主函数。

在预处理部分，首先引入需要的头文件：“stdio.h”和“string.h”。定义需要使用的全局变量：num_lines，代表当前指针yytext所在行数；num_chars，代表当前指针yytext所在列数；in_comments，代表当前指针yytext是否在多行注释中；in_one_comments，代表当前指针yytext是否在单行注释中。然后，定义函数void print(int line, int chars, char type)方便格式化输出。

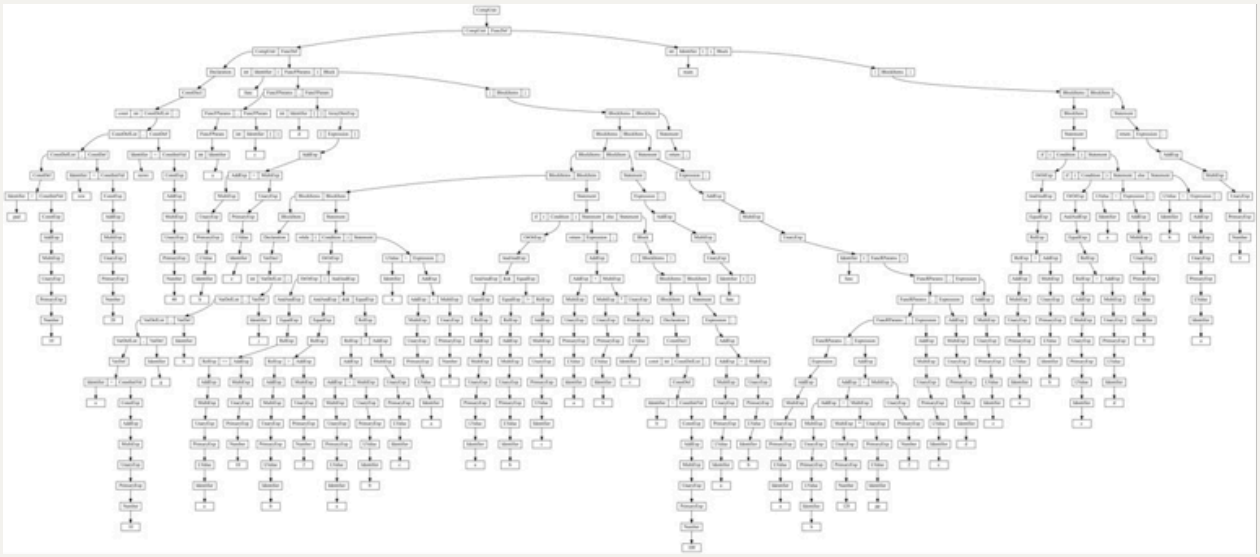
之后，定义方便表意的常量：Newline表示新一行，并针对Windows和Unix&Linux系统的两类换行CRLF和LF分别设计；Keyword表示SYSY语言关键字；Operator表示SYSY语言定义的运算符；Digit(n)表示10进制（或n进制）的数字字符；Integer(n)表示10进制（或n进制)的整数；InstantNum表示SysY语言所定义的立即数，即10进制、8进制、16进制整数的全体；Letter表示大小写字母；Identifier表示SysY语言定义的标识符；Delimeter表示SysY语言定义的界符；Blank表示非换行分隔符块。

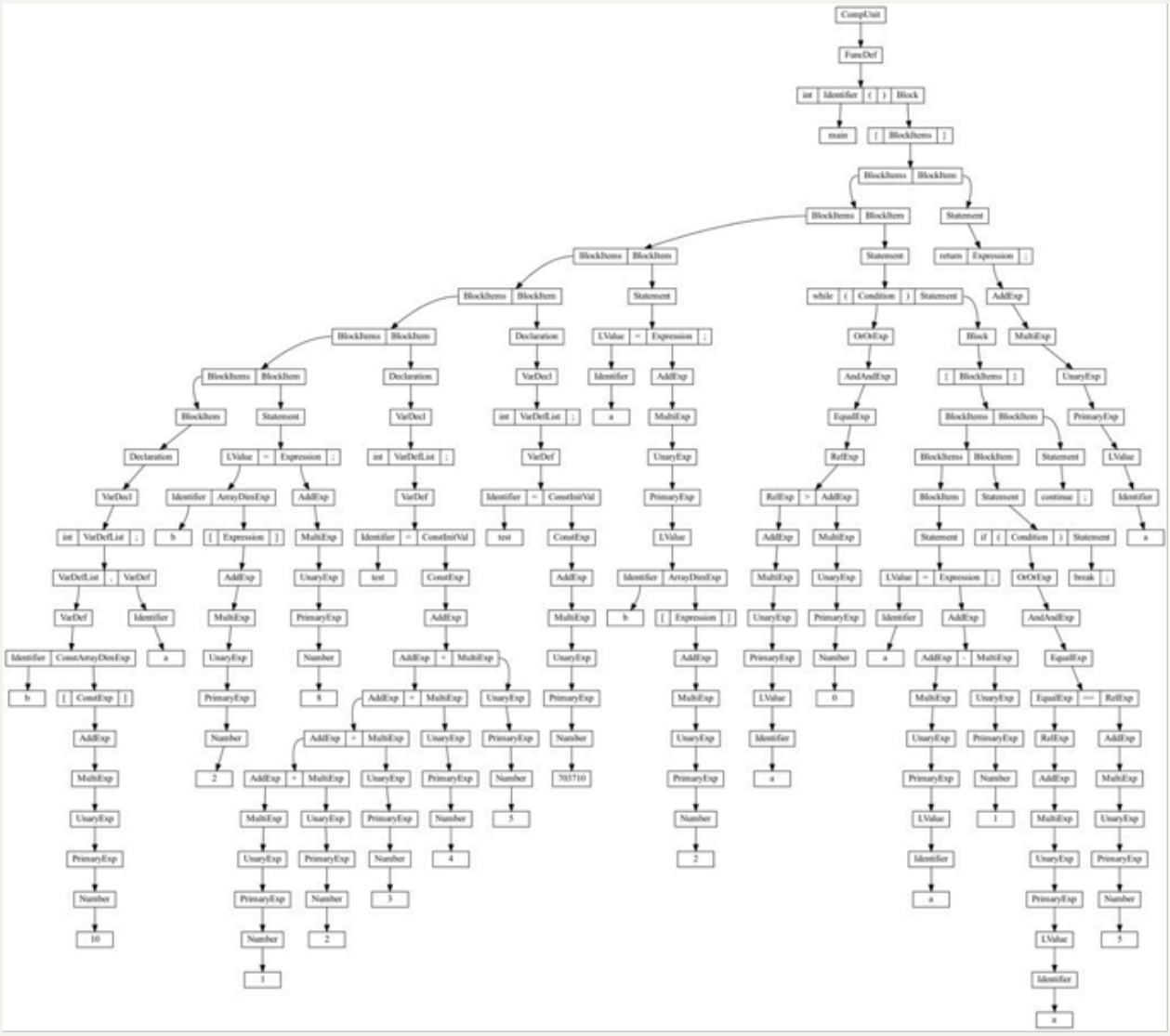
然后，定义扫描行为。对于单行注释开头“//”和多行注释开头“/”、*结尾*“/”，改变全局状态in_comments和in_one_comments的值，并更新行、列号；对于其余定义的表意单词类型，按“内容：类别（行号，列号）”的格式输出，并更新行、列号；对于空白块，更新行、列号；对于其余未定义字符，输出未定义字符。

最后，定义主函数。为方便使用，设置命令行参数。利用stdin重定向，可执行文件可以含SysY语言源代码的地址作为命令行参数，从而方便使用。调用yylex()。

3 语法分析部分

本部分基于SysY语言，在之前的LEX词法分析器基础上，对SysY语言文件进行语法分析，并利用Graphviz生成语法树图片。得出的语法树如下图组所示





在对上述语法树进行检验时，发现对于以下的要点均取得了正确的结果：

- 整体分析通过；
- 可以针对性识别运算符优先级、结合性；
- 可以满足if...else...语句的最近优先结合性；
- 可以识别八进制、十进制、十六进制整数；
- 可以识别多维度数组；

因此，本部分完成了需要完成的工作，可以为接下来的语义分析部分继续使用。

3.1 语法分析过程

本部分的分析过程如下：

1. 词法分析模块（LEX）对给定的文件进行词法分析，然后根据不同单词类型，创建树节点，并将其对应信息传递给yylval，返回终结符识别号；
2. 语法分析模块（YACC）根据词法分析模块拿到的终结符识别号决定移进/规约。如果规约，则新建树节点，并将其规约内容加入其孩子域中；
3. 如果开始符被规约，那么分析成功，并且输出为Graphviz树文件，并作图；
4. 如果出现了句法错误（Syntax Error），则终止程序，停止分析。

3.2 YACC脚本设计

YACC脚本主要分为以下的部分：

- C语言预处理部分和相关函数的定义；
- YACC标识符（%token和%node等）的定义；
- YACC语法的定义和语义动作的定义；
- 主函数（程序入口）的定义。

其中，YACC语法主要为LR规约形成预测分析表，用于进一步的分析使用。语义动作用于构造语法树，并为后续的部分扩展，最终形成可生成汇编码的小型SysY语言编译器。

4 语义分析部分

本部分基于词法分析器、语法分析器，设计汇编代码构建框架的语义动作和相关结构，最终生成能够为x86_64机器正常编译通过的AT&T汇编语句。

4.1 完成度说明

目前，已经实现的功能有：

- 主函数程序入口
- 主函数正常返回
- 普通和数组变量的声明、定义、访问和赋值
- 普通类型常变量的声明、定义和访问
- 全局普通类型和数组类型变量的声明、定义、访问和赋值
- 二元运算（+、-、*）
- 逻辑运算（&&、||、!）
- if语句块和if...else...语句块
- while循环语句块
- break和continue语句
- 变量嵌套定义和变量作用域选择
- 寄存器贪心方法分配
- 函数调用
- printf和scanf输入输出

暂不支持的功能有：

- 除法运算和括号
- 指针操作（如：`scanf(a[1])`）
- 列表初始化（如：`int a[] = {...};`）
- 非简单常左值或数字的数组定义（如：`int a[N * 2]`）
- 函数返回语句缺失或错误的处理
- 多层含局部变量的while循环语句的连续break、continue操作
- 其他在测试中出现错误的情况

4.2 整体设计

4.2.1 节点设计

定义每一个语法节点类型为Node，详细定义如下：

```
typedef struct Node
{
    Type type;
    char name[65];
    int value;

    int isNum;
    char idname[65];
    int idvalue;
    int dim;
    int dim1, dim2;
    int dim1type, dim2type;
    int offset;

    int truelist[16];
    int falselist[16];
    int truelen = 0, falselen = 0;
    int isneg = 0;

    struct Node* father;
    int child_cnt;
    struct Node* children[CHILD_CNT];
} Node;
```

其中，每个字段的含义如下：

- type：该节点的类型（非终结符类型对应非终结符的名称，终结符类型由词法分析器给出）
- name：该节点的语法树中的名称（若有）
- value：该叶子节点的值（若有）

- isNum：该非终结节点对应的信息类型
    - isNum为0，表示普通变量或普通常量，标识符名称由idname储存
    - isNum为1，表示立即数（已经转化为10进制），数值由idvalue储存
    - isNum为2，表示寄存器，寄存器编号（即枚举的字面值）由idvalue储存
    - isNum为3，表示数组类型，标识符名称由idname储存，维数由dim储存，每一维度的索引由dim1、dim2储存（暂时只支持2维），每一维度的索引值的类型由dim1type和dim2type储存（类型含义同isNum，或为立即数，如`a[1]`，或为寄存器编号，运算结果储存在寄存器中，如`a[m*n]`）。计算后的偏移由offset储存。
    - isNum为4，表示bool类型，真、假链及其长度分别由truelist&truelen和falselist&falselen储存。是否取反由isneg储存。
- father、children：语法树上该节点的双亲和孩子节点
- child_cnt：该节点的孩子节点个数

4.2.2 相关结构与函数设计

程序自定义Variable和Function结构体，分别表示某个特定的变量或者函数。并使用`std::vector<std::map <std::string, Variable> > varList`和`std::map <std::string, Function> funcList`表示符号表栈和函数表。

```
struct Variable
{
    // int type var
    string name;
    VarType type;
    int value;
    int pos; // pos(%rbp) or pos(%rbp, offset, 4)
    int dim;
    int dim1, dim2;

    Variable() = default;
    Variable(char * n, VarType t, int v, int d);
    Variable(char * n, VarType t);
};

struct Function
```

```
{
    string name;
    int hasReturnValue;
    vector <Variable> params;

    Function(string n);
    Function(const Function & other);
    Function() = default;
};
```

程序定义了汇编码生成器Generator类，具体定义如下。本类有全局对象，用于管理并生成汇编码。

```
class Generator
{
public:
    std::string path;
    std::ofstream fout;
    stringstream buffer;
    int nextAsm = 0;
    int nextLabel = 0;
    vector <string> asms;
    map <int, vector <string> > labels;
    int stackPos = 0; // 表示当前分配变量的栈位置

    Generator() = default;
    int newInt();
    int newBytes(int size);
    void genConst(Variable var);
    void genFuncHead(string name);
    void clearStackPos();
    bool open(std::string a);
    void close();
    void write(std::string content);
    void solidate();
    Generator &operator<<(std::string a)
};
```

程序使用到的全局函数如下，含义如名称：

```
void addNode(Node * father, Node * a, Node * b = NULL, Node * c = NULL, Node * d = NULL, Node * e = NULL);
Node * newNode(Type type);
void appendNode(Node * father, Node * son);
void generateGraph(Node * root, const char * a);
int insertVarList(char * name, VarType type, int value = NULL, int dim = NULL, int dim1 = NULL, int dim2 = NULL);
void setGenerator(char * p);
void genFuncHead(char * p);
void genReturn();
void freeReg(Registers r);
void setReg(Registers r);
Registers getFreeReg();
Registers transferToReg(char * varName);
Registers transferToReg(int num);
Registers arrayTransferToRegAndFree(char * arrayName, int offsetReg);
Registers genAdd(Node * a, Node * b);
Registers genMinus(Node * a, Node * b);
Registers genMulti(Node * a, Node * b);
void genAssign(Node * a, Node * b);
int findConstValue(Node * node, int * valid);
int calcOffsetAndFreeReg(Node * nameNode, Node * dimNode, int * type);
void solidate();
int getNextI();
void out(Node * a); // 跳出的无条件跳转
int genJmp();
void testZero(Node * a);
void genTrueFalseZero(Node * a, int isneg = NULL); // 在比较后使用，仅仅生成两个链
void testCompare(Node * a, Node * b);
void genTrueFalseCompare(Node * a, char type, int equal = NULL);
void modifyAsm(int line, int target, int isbreak = NULL); //把第line行回填上target行，判断是否是break
void cpylist(Node * tar, Node * from);
```

```
int mergelist(int * des, int deslen, int * from, int fromlen); // des = des 并 from
void backpatch(Node * a, int truetarget = -1, int falsetarget = -1);
void genReturn(Node * a);
void insertFuncList(char * name);
void addParam(char * funcName, char * varName, VarType type, int dim = NULL, int dimContent = NULL);
void processParams(char * funcName);
void call(char * funcName);
//void popParams(char * funcName);
void addToWaitPush(Node * node); // 采取等待队列的方法
void pushParams();
void popVarList(); // 弹出符号表
void genPrintf(Node * a);
void genScanf(Node * a);
void enterStmt();
void exitStmt();
```

5 文件说明

本项目文件文件结构如下：

```
.
├── Build
|   ├── 0
|   ├── 1
|   ├── 2
|   ├── 3
|   └── 4
├── Generator.hpp
├── grammar.y
├── lex.yy.c
├── Makefile
├── Output
|   ├── 0.s
|   ├── 1.s
|   ├── 2.s
```

```
|   ├── 3.s
|   ├── 4.s
├── readme.md
├── show.sh
├── sysyc
├── Test
|   ├── 0.sy
|   ├── 1.sy
|   ├── 2.sy
|   ├── 3.sy
|   ├── 4.sy
├── tree.cpp
├── tree.h
├── wordAnalysis.l
├── y.tab.c
└── y.tab.h
```

- y.tab.*为YACC生成的C语言源文件和头文件
- lex.yy.c为FLEX生成的C语言源文件
- Makefile为项目make文件，其中，main目标为本项目的生成
- show.sh为分别编译五个测试样例为汇编语言，并用gcc编译生成Build目录下的可执行文件的shell脚本
- sysyc为本项目生成的编译器可执行二进制文件
- Generator.hpp、grammar.y、tree.h/cpp、wordAnalysis.l为本项目的源文件和头文件。

[注意] 项目文件中的3.sy在Linux Ubuntu 20.04上由于未知错误，无法生成3.s。这里的3.s是在macOS 12.4中生成的；可执行文件3是在Linux Ubuntu 20.04上根据macOS生成的3.s文件使用gcc编译、链接产生的。

附录A 为测试而编写的分功能测试样例

运算测试

```
const int a = 1;

int main()
{
    int a = 2, b = 0x11;
    int c = 3+ a * b;
    int d = a*2+3;
    int e = a*b-c*c;
    int f = a*c-d*e+1;
    return 0;
}
```

数组测试

```
const int N = 10;

int main()
{
    int a[N][N];
    a[2*3][3*2] = 2*2+2*a[2][2];
    return 0;
}
```

条件分支测试

```
const int N = 100;
int main()
{
    int a = 2;
    if(1+1 == 2*a+1)
    {
        a = 1;
        a = a * 2;
    }
  return 0;
}
```

布尔表达式测试

```
const int N = 100;
int main()
{
    int a = 2, b[2][3];

    if(a > 1 && a > 2 || a > 3)
    {
        a = 1;
    }
    else b[1][1] = 2;
}
```

循环测试

```
const int N = 10;
int main()
{
    int i = 0, sum = 0;
    while(1)
    {
        if(i == 0)
        {
            i = i + 1;
            continue;
        }
        if(i > N)
            break;
        else
        {
            sum = sum + i;
            i = i + 1;
        }
    }
    return sum;
}
```

函数调用测试

```
int calc_sum(int N)
{
    int i = 0, sum = 0;
    while(1)
    {
        if(i == 0)
        {
            i = i + 1;
            continue;
        }
        if(i > N)
            break;
        else
        {
            sum = sum + i;
            i = i + 1;
        }
    }

    return sum;
}

int main()
{
    int x;
    scanf(x);
    printf(calc_sum(x));
    return 0;
}
```

全局变量测试

```
int a = 2, b = 3;
int M[2][2];

int main()
```

```
{
    scanf(b);

    int a[2];
    a[1] = b*2;
    b = b + 2;
    M[1][1] = b+3;
    int p = b + M[1][1] + a[1];
    printf(p);
    return 0;
}
```

经测试，以上测试内容均可获得正确的结果。