南 开 大 学

Nankai University

编译系统原理实验报告

REPORT FOR COMPILERS PRINCIPLES

定义语言特性及汇编编程

学 科 专 业： 计 算 机 科 学 与 技 术

学 号： 1911463 1911433

作 者 姓 名： 时 浩 铭 林 坤

指 导 老 师： 王 刚

摘 要

本次实验确立了后续实验所要实现的编译器所支持的 SysY 语言特性，并设计出了上下文无关文法。此外，还简单探究了 SysY 运行时库的使用。

由于最终要编译出 ARM 平台上的可执行代码，因此此次实验也完成了对于 ARM 汇编的初次了解，手动编写了与 C++ 程序等价的 ARM 汇编程序，为后续实验打下了基础。

此外，还简单探究了实现编译 SysY 程序的编译器程序的数据结构和算法的设计。

关键词：编译器、SysY、ARM

目 录

第 1 章 引言

本次实验确定了后期所要编写的编译器支持的 SysY 语言特性，并根据 SysY 中巴克斯瑙尔范式定义，用上下文无关文法描述了要支持的语言特性。此外，实验还根据要支持的语言特性，设计了两个 SysY 程序，手动编写了等价的 ARM 汇编程序。

本次实验为小组合作完成，小组成员有时浩铭 (1911463) 与林坤 (1911433)。在此次实验中，两人各负责了一些上下文无关文法的设计，例如终结符号集合、非终结符号集合、开始符号以及部分产生式由时浩铭负责，其余产生式及运行时库由林坤负责。至于汇编程序的编写，时浩铭负责求素数程序的编写，林坤负责求斐波那契数列前 20 项的程序的编写。最后的思考部分，由二人共同完成。

本次实验在 Ubuntu 系统（虚拟机/WSL）下进行，具体设备的软硬件信息如下所示：

- 操作系统：Ubuntu 20.04.3 LTS
- CPU: Intel(R) Core(TM) i7-9750H CPU @2.60GHz 6 核 12 线程
- 内存: 16.0GB
- 交叉编译器: gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1 20.04)

我们项目的 gitlab 地址：https://gitlab.eduxiji.net/nku2021-lknb/compile/-/tree/master/arm

第 2 章 定义语言特性

2.1 概要

该语言类似 SysY 语言，是 C++ 语言的一个子集。程序源代码存储在扩展名为 sy 的文件中，每个 sy 文件有且仅有一个名为 main 的主函数定义，还可以包含全局变量声明、常量声明与其他函数定义。该语言支持 int 类型及 int 类型行有限存储的数组，支持调用运行时库。

计划完成要求的 8 个模块的所有内容，包括 int 数据类型及数组、变量常量、语句、表达式、注释、输入输出、函数和代码优化等内容。

2.2 语言文法

使用上下文无关文法来描述支持的预言特性。对于上下文无关文法 (V_T, V_N, P, S)，下面依次给出定义。

2.2.1 终结符号集合 V_T

终结符是由单引号引起的字符串或者是标识符 (Ident) 和数值常量 (IntConst)。

对于标识符，与 C 语言类似，具体见下上下文无关文法。

$$
\begin{aligned}
\text{identifier} \rightarrow\ & \text{identifier\_nondigit} \\
& \mid \text{identifier}\ \ \text{identifier\_nondigt} \\
& \mid \text{identifier}\ \ \text{digit} \\
\text{identifier\_nondigit} \rightarrow\ & \text{\_} \mid \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m} \\
& \mid \text{n} \mid \text{o} \mid \text{p} \mid \text{q} \mid \text{r} \mid \text{s} \mid \text{t} \mid \text{u} \mid \text{v} \mid \text{w} \mid \text{x} \mid \text{y} \mid \text{z} \\
& \mid \text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E} \mid \text{F} \mid \text{G} \mid \text{H} \mid \text{I} \mid \text{J} \mid \text{K} \mid \text{L} \mid \text{M} \\
& \mid \text{N} \mid \text{O} \mid \text{P} \mid \text{Q} \mid \text{R} \mid \text{S} \mid \text{T} \mid \text{U} \mid \text{V} \mid \text{W} \mid \text{X} \mid \text{Y} \mid \text{Z} \\
\text{digit} \rightarrow\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

关于同名标识符，允许全局变量和局部变量同名，在二者作用域重叠部分局部变量优先，不允许局部变量同名。此外，允许变量与函数同名。

对于数值常量，支持八进制、十进制和十六进制。具体如下文法：

$$
\begin{aligned}
\text{integer\_const} \rightarrow\ & \text{decimal\_const} \\
& \mid \text{octal\_const} \\
& \mid \text{hexadecimal\_const}
\end{aligned}
$$

$$
\begin{aligned}
\text{decimal\_const} &\rightarrow \text{nonzero\_digit} \\
&\quad \mid \text{decimal\_const}\ \ \text{digit} \\
\text{octal\_const} &\rightarrow 0 \mid \text{octal\_const}\ \ \text{octal\_digit} \\
\text{hexadecimal\_const} &\rightarrow \text{hexadecimal\_prefix}\ \ \text{hexadecimal\_digit} \\
&\quad \mid \text{hexadecimal\_const}\ \ \text{hexadecimal\_digit} \\
\text{hexadecimal\_prefix} &\rightarrow \text{`0x'} \mid \text{`0X'} \\
\text{nonzero\_digit} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\text{octal\_digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \\
\text{octal\_digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
&\quad \mid \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid \text{f} \mid \text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E} \mid \text{F}
\end{aligned}
$$

2.2.2 非终结符号集合 V_N

根据 SysY 语言定义文件，有以下非终结符。

- 编译单元 CompUnit
- 声明 Decl
- 常量声明 ConstDecl
- 基本类型 BType
- 常数定义 ConstDef
- 常量初值 ConstInitVal
- 变量声明 VarDecl
- 变量定义 VarDef
- 变量初值 InitVal
- 函数定义 FuncDef
- 函数类型 FuncType
- 函数形参表 FuncFParams
- 函数形参 FuncFParam
- 语句块 Block
- 语句块项 BlockItem
- 语句 Stmt
- 表达式 Exp
- 条件表达式 Cond
- 左值表达式 LVal
- 基本表达式 PrimaryExp
- 数值 Number
- 一元表达式 UnaryExp
- 单目运算符 UnaryOp
- 函数实参表 FuncRParams
- 乘除模表达式 MulExp
- 加减表达式 AddExp
- 关系表达式 RelExp
- 相等性表达式 EqExp
- 逻辑与表达式 LAndExp
- 逻辑或表达式 LOrExp
- 常量表达式 ConstExp

2.2.3 开始符号 S

开始符号为编译单元 CompUnit。

2.2.4 产生式集合 P

- 编译单元 CompUnit

$$\begin{aligned}\text{CompUnit} \rightarrow\ & \text{CompUnit}\ \ \text{Decl} \\ |\ & \text{CompUnit}\ \ \text{FuncDef} \\ |\ & \text{Decl} \\ |\ & \text{FuncDef}\end{aligned}$$

程序由单个文件组成，每个文件内容即为一个 CompUnit。

- 声明 Decl

$$\text{Decl} \rightarrow\ \ \text{ConstDecl}\ \text{‘;’} \mid \text{VarDecl}\ \text{‘;’}$$

- 常量与变量该部分中补充了一些非终结符从而完成上下文无关文法的定义，下面部分亦然。

$$\begin{aligned}\text{ConstDecl} \rightarrow\ & \text{‘}\textbf{const}\text{’}\ \ \text{BType}\ \ \text{ConstDef} \\ |\ & \text{ConstDecl}\ \ \text{‘,’}\ \ \text{ConstDef} \\ \text{BType} \rightarrow\ & \text{‘}\textbf{int}\text{’} \\ \text{ConstDef} \rightarrow\ & \textbf{Ident}\ \ \text{ArrayIndices}\ \ \text{‘=’}\ \ \text{ConstInitVal} \\ |\ & \textbf{Ident}\ \ \text{‘=’}\ \ \text{ConstInitVal} \\ \text{ConstArrayIndices} \rightarrow\ & \text{ConstArrayIndex} \\ |\ & \text{ConstArrayIndices}\ \text{‘,’}\ \text{ConstArrayIndex} \\ \text{ConstArrayIndex} \rightarrow\ & \text{‘[’}\ \text{ConstExp}\ \text{‘]’} \\ \text{ConstInitVal} \rightarrow\ & \text{ConstExp} \\ |\ & \text{‘\{’}\ \text{‘\}’} \\ |\ & \text{‘\{’}\ \text{ConstInitValList}\ \text{‘\}’} \\ \text{ConstInitValList} \rightarrow\ & \text{ConstExp} \mid \text{ConstInitValList}\ \text{‘,’}\ \text{ConstExp} \\ \text{VarDecl} \rightarrow\ & \text{BType}\ \ \text{VarDef} \\ |\ & \text{VarDecl}\ \text{‘,’}\ \text{VarDef} \\ \text{VarDef} \rightarrow\ & \textbf{Ident} \\ |\ & \textbf{Ident}\ \ \text{ArrayIndices} \\ |\ & \textbf{Ident}\ \ \text{‘=’}\ \ \text{InitVal} \\ |\ & \textbf{Ident}\ \ \text{ArrayIndices}\ \ \text{‘=’}\ \ \text{InitVal}\end{aligned}$$

$$
\begin{aligned}
\text{InitVal} \rightarrow\ & \text{Exp} \\
| \ & \text{`\{' `\}'} \\
| \ & \text{`\{' InitValList `\}'} \\
\text{InitValList} \rightarrow\ & \text{Exp} \mid \text{InitValList `,' Exp}
\end{aligned}
$$

- 函数

$$
\begin{aligned}
\text{FuncDef} \rightarrow\ & \text{FuncType } \textbf{Ident} \text{ `(' `)'Block} \\
| \ & \text{FuncType } \textbf{Ident} \text{ `(' FuncFParams `)'Block} \\
\text{FuncType} \rightarrow\ & \text{`}\textbf{void}\text{'} \mid \text{`}\textbf{int}\text{'} \\
\text{FuncFParams} \rightarrow\ & \text{FuncFParam} \mid \text{FuncFParams`,'FuncFParam} \\
\text{FuncFParam} \rightarrow\ & \text{BType } \textbf{Ident} \mid \text{BType } \textbf{Ident}\text{`[' `]'} \\
| \ & \text{FuncFParam ArrayIndices} \\
\text{ArrayIndices} \rightarrow\ & \text{ArrayIndex} \\
| \ & \text{ArrayIndices `,' ArrayIndex} \\
\text{ArrayIndex} \rightarrow\ & \text{`[' Exp `]'} \\
\text{FuncRParams} \rightarrow\ & \text{Exp} \mid \text{FuncRParams `,' Exp}
\end{aligned}
$$

- 语句

$$
\begin{aligned}
\text{Block} \rightarrow\ & \text{`\{'BlockItems`\}'} \mid \text{`\{' `\}'} \\
\text{BlockItems} \rightarrow\ & \text{BlockItem} \mid \text{BlockItems BlockItem} \\
\text{BlockItem} \rightarrow\ & \text{Decl} \mid \text{Stmt} \\
\text{Stmt} \rightarrow\ & \text{`;'} \mid \text{Exp `;'} \\
| \ & \text{LVal `=' Exp `;'} \mid \text{Block} \\
| \ & \text{`}\textbf{if}\text{' `(' Cond `)' Stmt} \\
| \ & \text{`}\textbf{if}\text{' `(' Cond `)' Stmt `}\textbf{else}\text{' Stmt} \\
| \ & \text{`}\textbf{while}\text{' `(' Cond `)' Stmt} \\
| \ & \text{`}\textbf{break}\text{' `;'} \\
| \ & \text{`}\textbf{continue}\text{' `;'} \\
| \ & \text{`}\textbf{return}\text{' `;'} \\
| \ & \text{`}\textbf{return}\text{' Exp `;'}
\end{aligned}
$$

- 表达式

$$
\text{Exp} \rightarrow \text{AddExp}
$$

$$
\begin{aligned}
\text{Cond} &\rightarrow \text{LOrExp} \\
\text{LVal} &\rightarrow \textbf{Ident}\ \ \text{ArrayIndices} \\
\text{PrimaryExp} &\rightarrow \text{`(' Exp `)'} \mid \text{LVal} \mid \text{Number} \\
\text{Number} &\rightarrow \textbf{IntConst} \\
\text{UnaryExp} &\rightarrow \text{PrimaryExp} \mid \\
&\quad \mid \textbf{Ident}\text{`(' `)'} \\
&\quad \mid \textbf{Ident}\text{`(' FuncRParams `)'} \\
&\quad \mid \text{UnaryOpUnaryExp} \\
\text{UnaryOp} &\rightarrow \text{`+'} \mid \text{`-'} \mid \text{`!'} \\
\text{MulExp} &\rightarrow \text{UnaryExp} \\
&\quad \mid \text{MulExp `*' UnaryExp} \\
&\quad \mid \text{MulExp `/' UnaryExp} \\
&\quad \mid \text{MulExp `\%' UnaryExp} \\
\text{AddExp} &\rightarrow \text{MulExp} \\
&\quad \mid \text{AddExp `+' MulExp} \\
&\quad \mid \text{AddExp `-' MulExp} \\
\text{RelExp} &\rightarrow \text{AddExp} \\
&\quad \mid \text{RelExp `<' AddExp} \\
&\quad \mid \text{RelExp `>' AddExp} \\
&\quad \mid \text{RelExp `<=' AddExp} \\
&\quad \mid \text{RelExp `>=' AddExp} \\
\text{EqExp} &\rightarrow \text{RelExp} \\
&\quad \mid \text{EqExp `==' RelExp} \\
&\quad \mid \text{EqExp `!=' RelExp} \\
\text{LAndExp} &\rightarrow \text{EqExp} \\
&\quad \mid \text{LAndExp `\&\&' EqExp} \\
\text{LOrExp} &\rightarrow \text{LAndExp} \\
&\quad \mid \text{LOrExp `||' LAndExp} \\
\text{ConstExp} &\rightarrow \text{AddExp}
\end{aligned}
$$

2.3 SysY 运行时库

SysY 运行时库提供一系列 I/O 函数、计时函数等用于在 SysY 程序中表达输入/输出、计时等功能需求。部分 SysY 库函数的参数类型会超出 Sys Y 支持的数据类型，如可以为字符串。SysY 编译器需要能处理这种情况，将参数正确地传递给 SysY 运行时库。

2.3.1 SysY 运行时库相关文件

libsysy.a 和 libsysy.so 分别是 SysY 运行时库的静态库和动态库文件。

sylib.h 其中包含 SysY 运行时库涉及的函数等的声明。

在 SysY 源程序中不出现对 sylib.h 的文件包含，而由 SysY 编译器来分析和处理 SysY 程序中对这些函数的调用。

2.3.2 I/O 函数

SysY 运行时库提供一系列 I/O 函数，支持对整数、字符以及一串整数的输入和输出。为便于在 SysY 程序中控制输出的格式，诸如 putf 这样的 I/O 函数会使用超出 Sys Y 语言支持的数据类型的参数，如格式字符串。

- `int getint()`
  输入一个整数，返回对应的整数值。
  示例：`int n; n = getint();`
- `int getch()`
  输入一个字符，返回字符对应的 ASCII 码值。
  示例：`int n; n = getch();`
- `int getarray(int [])`
  输入一串整数，第 1 个整数代表后续要输入的整数个数，该个数通过返回值返回；后续的整数通过传入的数组参数返回。
  注：getarray 函数获取传入的数组的起始地址，不检查调用者提供的数组是否有足够的空间容纳输入的一串整数。
  示例：`int a[10][10]; int n; n = getarray(a[0]);`
- `void putint(int)`
  输出一个整数的值。
  示例：`int n=10; putint(n); putint(10); putint(n);`
  将输出: 101010
- `void putch(int)`
  将整数参数的值作为 ASCII 码，输出该 ASCII 码对应的字符。

注：传入的整数参数取值范围为 0 255，putch()不检查参数的合法性。

示例：int n=10; putch(n);

将输出换行符

- void putarray(int, int[])

  第 1 个参数表示要输出的整数个数 (假设为 N)，后面应该跟上要输出的 N 个整数的数组。putarray 在输出时会在整数之间安插空格。

  注：putarray 函数不检查参数的合法性。

  示例：int n=2; int a[]={2,3}; putarray(n, a);

  输出：2: 2 3

- void putf (<格式串>, int, …)

  第 1 个参数为格式字符串，其中仅包含 2 种格式符，即‘%d’和‘%c’；该函数将根据格式串进行输出，遇到普通字符则原样输出，遇到格式符‘%d’或‘%c’则从第 2 个参数起依次取对应参数的值按整数或字符输出。

  示例：int n=2; int a[]={2,3};

  putf(“%d: %d(%c), %d(%c)”, n, a[0], a[0]+48, a[1], a[1]+48);

  输出：2: 2(2), 3(3)

2.3.3 计时函数

SysY 运行时库提供 starttime、stoptime 一对函数用于对 SysY 中的一段代码的运行进行计时。在一个 SysY 程序中，可以插入多对 starttime、stoptime 调用来获得每对调用之间的代码的执行时长，并在 SysY 程序执行结束后得到这需要注意的是：starttime、stoptime 不支持嵌套调用的形式，即不支持 starttime()…starttime()…stoptime()…stoptime()这样的调用执行序列。

计时函数使用示例

```
1  void foo(int n){
2    starttime();
3    for(int i=0;i<n;i++)
4      system(”sleep 1”);
5    stoptime();
6  }
7  int main(){
8    starttime();
9    for(int i=0;i<3;i++)
10     system(”sleep 1”);
11   stoptime();
12   foo(2);
```

```
13 }
```

计时函数使用示例输出

```
1 Timer#001@0010-0012: 0H-0M-3S-3860us
2 Timer#002@0005-0007: 0H-0M-2S-2660us
3 TOTAL: 0H-0M-5S-6520us
```

第 3 章 汇编编程

实验采用 ARM 汇编，具体实验步骤为首先编写 C++ 程序，然后编写汇编程序，得到可以正确执行的可执行结果，然后再与交叉编译器生成的汇编代码进行比较。

分别编写了求素数和求斐波那契数列两个程序，具体实验代码及分析如下。

3.1 求素数程序

首先给出程序的 SysY 代码。

求素数 SysY 代码

```
#include <cmath>

int isPrime(int n) {
    if (n < 2)
        return 0;
    if (n == 2)
        return 1;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) {
            return 0;
        }
    }
    return 1;
}

int main() {
    int n;
    n = getint();
    for (int i = 1; i < n; i++) {
        if (isPrime(i))
            putf("%d ", i);
    }
}
```

该程序从键盘读入一个整数 n，然后输出以空格分隔的 1-n 中的素数，最后打印换行。该程序包含了函数、局部变量、循环、表达式等语言特性。

编写该程序时，遇到了较多的困难，例如该程序涉及到了整数的平方根，也就不可避免的涉及到了小数与整数的转换，需要使用浮点协处理器来进行小数的运算。此外就

是汇编程序对于段的分隔较为严格，需要用到.align 进行分隔，否则也会产生较多错误。

最终得到的代码如下：

求素数 ARM 代码

```
    .arch armv8-a

    .comm n, 4
    .text
    .align 2
    .section .rodata
    .align 2
str:
    .ascii "%d \00"
str1:
    .ascii "\n\00"
str2:
    .ascii "%d\00"

    .text
    .align 1
    .global sqrt_int
    .syntax unified
    .fpu vfpv3-d16
sqrt_int:
    push {lr}
    vmov s15, r1
    vcvt.f64.s32 d7, s15
    vmov.f64 d0, d7
    bl sqrt
    vcvt.s32.f64 s15, d0
    vmov r2, s15
    pop {pc}

    .global isPrime
    .syntax unified
    .fpu vfpv3-d16
isPrime:
    @ r1 n
    @ r3 i
```

```
36      @ r2 sqrt(n)
37      push {r4, r5, lr}
38      cmp r1, #2
39      beq true
40      blt false
41      bl sqrt_int
42      mov r3, #2
43  L2:
44      cmp r3, r2
45      bgt true
46      sdiv r4, r1, r3
47      mul r5, r3, r4
48      mov r4, r1
49      sub r4, r4, r5
50      cmp r4, #0
51      beq false
52      add r3, r3, #1
53      b L2
54
55  false:
56      mov r0, 0
57      b ret
58  true:
59      mov r0, 1
60  ret:
61      pop {r4, r5, pc}
62
63      .global main
64      .syntax unified
65            .fpu vfpv3-d16
66  main:
67      push {r4, r5, fp, lr}
68      ldr r0, _bridge+8
69      ldr r1, _bridge+12
70      bl __isoc99_scanf
71      ldr r3, _bridge+12
72      ldr r5, [r3]
73      mov r4, #1
74  L1:
75      mov r1, r4
```

```
    bl isPrime
    cmp r0, #1
    bne not
    ldr r0, _bridge
    mov r1, r4
    bl printf
not:
    add r4, r4, #1
    cmp r4, r5
    bgt L1_END
    b L1

L1_END:
    ldr r0, _bridge+4
    bl printf
    mov r0, #0
    pop {r4, r5, fp, pc}

_bridge:
    .word str
    .word str1
    .word str2
    .word n

    .section .note.GNU-stack,"",%progbits
```

与编译器得到的汇编代码相比，编译器的代码更长，会加入很多编译器标记。代码流程上，主体保持一致。寻址方式上，编译器采用的更复杂，同时也会加入一些检查错误的代码。编译器会添加许多的代码标记，把代码分为几个部分。具体是如何分开的，由于标记并没有意义，因此也并不能完全理解。

3.2 求斐波那契数列程序

打印斐波那契数列的 c++ 代码。

求斐波那契数列 SysY 代码

```
#include<stdio.h>
using namespace std;
void fiboprint(int n);
int n=20;
int fib[20];
```

```
int main(){
    fiboprint(n);
    return 0;
}

//打印斐波那契数列前n项
void fiboprint(int n){
    int* fib = new int[n];
    fib[1] = 1;
    fib[0] = 1;
    for (int i = 0;i < n;i++){
        if (i == 0 || i == 1)
            printf("%d ", fib[i]);
        else{
            fib[i] = fib[i - 1] + fib[i - 2];
            printf("%d ", fib[i]);
        }
    }
    return;
}
```

该程序实现了在内存中开辟空间定义全局数组 fib, 用于存放斐波那契数列中的每一项值，同时通过调用函数求解斐波那契数列的每项值，并在函数中将求出的斐波那契数列值打印。

编写对应的 arm 汇编程序时需要注意对代码进行分段，arm 汇编代码需要分为数据段 (.data) 以及代码段 (.text)，同时还可将一些常量数据放置于只读数据段 (.rodata)，例如用于辅助打印的字符串。数组标号在声明的同时一定要注意利用.space为数组预留足够的内存空间，否则会引发未知的错误。在访问内存中的数据时要注意ldr指令所取的值究竟是地值址还是对应地址上的值。最后函数调用时注意寄存器值恢复即可。

求斐波那契数列 arm 汇编代码

```
    .data
n:
    .word   20

    .global array
    .bss
    .align 2
    .size array, 80
array:
```

```
    .space 80

@常量区
    .section .rodata
    .align 2
str0:
    .ascii "%d \0"

    .global main
    .global fibprint

    .text

main:
    push {lr}
    bl fibprint
    pop {pc}

fibprint:
    push {lr}
    @r0存储项数,r1存array的地址
    @r4 计数器
    ldr r11,_bridge+8
    mov r4,#0
    @r5 中间变量计算结果
    mov r5,#1
    @中间变量计算结果放入对应的地址中
    str r5,[r11,r4,LSL #2]
    add r4,r4,#1
    str r5,[r11,r4,LSL #2]
    mov r4,#0


LOOP:
    mov r9,#1
    cmp r4,#0
    beq L1
    cmp r4,#1
    beq L1
    sub r5,r4,#1
```

```
50      sub r6,r4,#2
51      ldr r7,[r11,r5,LSL #2]
52      ldr r8,[r11,r6,LSL #2]
53      add r9,r7,r8
54      str r9,[r11,r4,LSL #2]
55      @打印
56
57  L1:
58      ldr r0,_bridge+4
59      mov r1,r9
60      bl printf
61      add r4,r4,#1
62      ldr r10,_bridge
63      ldr r10,[r10]
64      cmp r4,r10
65      beq END
66      b LOOP
67  END:
68
69      pop {pc}
70
71  _bridge:
72      .word   n
73      .word   str0
74      .word   array
```

第 4 章　编译器程序的数据结构和算法设计

4.1　词法分析

最开始的时候，高级语言编写的程序对编译器来说只是一连串的单个字符组成的字符串。为了让编译器识别这一连串的字符串，需要逐个字符的读取源程序，然后将其切分成有意义的单词，这些被切分后的单词在编译器眼里是以 < 标识，语义值 > 对的形式存在。为了从源程序字符串中依次找出单词，编译器需要具有扫描功能，通常这种扫描器可以用一组有限状态机来实现。

4.1.1　有限状态机

下图为一个识别数字的有限状态机，数字由整数部分和可选的小数部分组成。因此，根据这个有限状态机，例如 200 和 2.58 都能被识别成一个有效的数字。

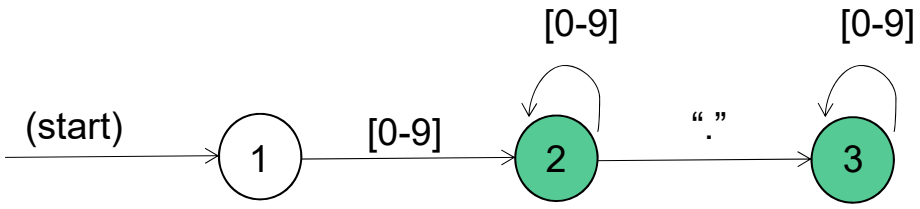

图 4.1　小数有限状态机

绿色的结点用环形标志，表示他们是“可接受”的状态，也就是说，只要我们的状态达到了绿色的结点，就表示我们之前读取到的数据是一个有效的数字。例如，从图中的 start 处开始，如果我们读到的数字是 42.15，那么依次经历的状态是（1,2,2,3,3,3），由于这一系列状态最终以“可接受”的状态结束 (也就是图中的状态 3)，因此我们就读取到了一个有效的数字。而且读取到的数字 42.15 用 < 标识，值 > 对的形式表示成 <NUMBER,42.15>。这里的 NUMBER 是用于标识我们读取到的内容是一个数字，而文本“42.15”是标识对应的语义元素值。

我们可以用为不同类型的单词（标识符，数字，关键字）定义不同的类似上述的小状态机，例如变量名可以由字母、下划线组成，操作符可以取 +=、->，关键字则可以是“if”“while”等单词，类型则可以是“int”“char”等等单词。为每一类单词构造一个小的有限状态机，最终组成一个可以接受不同类型单词的大状态机。可以用表的形式存放我们得到的大状态机。至此，我们通过构造一个大状态机得到一个能识别各类单词的自动扫描器。

4.2 语法分析

完成词法分析后则需要依据所得到的词构建对应的语法分析树。

4.2.1 语法分析树

进行语法分析时则需要用到一种数据结构树，此处生成的树称为语法分析树，对于下列语句：

语法分析树语句

```
if ( x == 2 )
{
    x = a + b;
}
```

生成语法分析树如下：

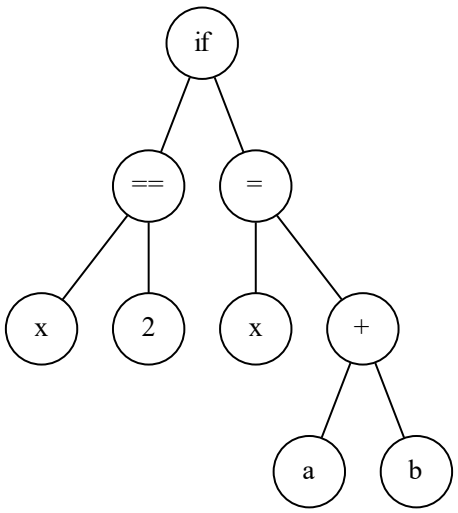

图 4.2 语法分析树

4.2.2 预测分析法

语法分析树可以利用预测分析法生成，具体做法是从一端扫描整个程序，用占位符为所有之前没遇到的元素创建一个临时语法树，然后依次读取后续的数据来填充完这颗语法树。例如上例的 if 语句，如果在扫描程序时，读取到了 if 这个单词，则首先初步判定其为 if 语句，临时生成的语法树为：

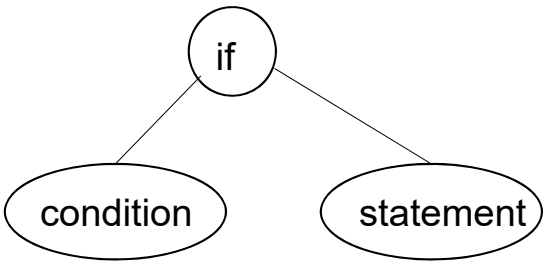

图 4.3 临时语法分析树

同时再不断读取接下来的程序，用以匹配 if 语句的结构不断完善语法分析树，若一旦不符合 if 语句的结构则会报出语法错误。同时会有一个对应函数 match 进行对应语句结构的匹配。condition 以及 statement 也可以进行相对应的匹配生成对应的语法分析子树，不断完善最终的语法分析树。

4.3 翻译语法树

有了语法树后，我们接下来要做的事情是构建符号表，以便确定各个元素在存储器中的存放位置。

具体做法：遍历语法树，将语法树中不同的变量依次取出，放入可用的存储位置。编译器自己决定如何分配存储位置。

具体过程可以用下图表描述：

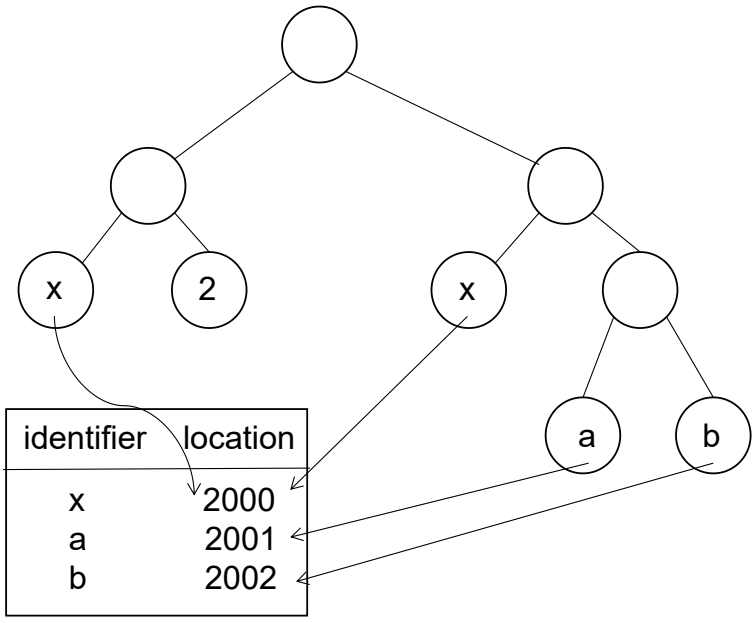

图 4.4 各变量地址分配

语法树中多次出现的变量指向同一地址。从上到下，从左往右依次遍历语法树，遇到一个 if 结点，执行 if 相关的操作，遇到赋值结点，执行赋值相关的操作，详细步骤如下所示：

首先，寻找最小表达式，如下图中绿色、蓝色圈中的即为一个最小表达式。

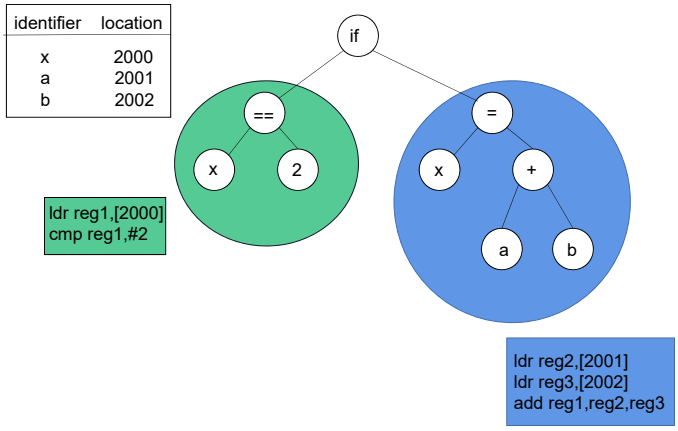

图 4.5 最小表达式生成

接下来，将最小表达式与其周边的表达式合并。最终获取最后的汇编语言。

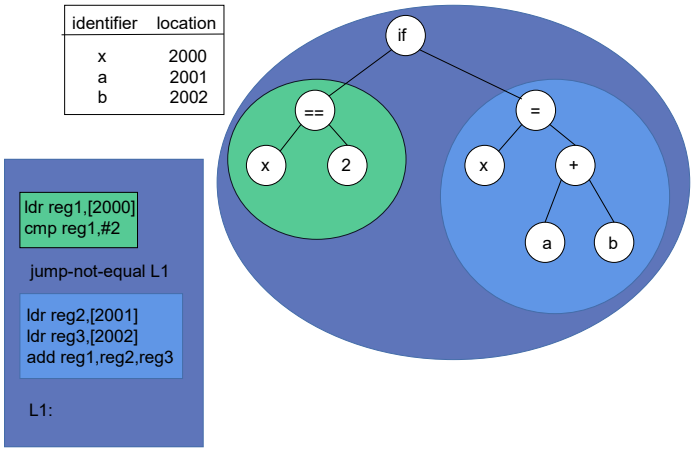

图 4.6 最终汇编语言生成

以上简单的介绍了 if 语句到汇编语句的具体转化翻译过程，其余的语句也可以据其对应结构生成语法分析树。可将编写好的源程序划分为多个不同的语句块，并依据每一个语句块构建起来的语法分析树得到对应语句块翻译出的汇编程序代码。最后再将每一个语句块按照顺序组合即可得到整个源程序对应的汇编程序代码，有能力实现将所有的高级语言翻译为汇编语言。