附录A：C−−语言文法

在本附录中，我们给出C−−语言的文法定义和补充说明。

文法定义

Tokens

```
INT → /* A sequence of digits without spaces[1] */
FLOAT → /* A real number consisting of digits and one decimal point. The decimal point must be surrounded by at least one digit[2] */
ID → /* A character string consisting of 52 upper- or lower-case alphabetic, 10 numeric and one underscore characters. Besides, an identifier must not start with a digit[3] */
SEMI → ;
COMMA → ,
ASSIGNOP → =
RELOP → > | < | >= | <= | == | !=
PLUS → +
MINUS → -
STAR → *
DIV → /
AND → &&
OR → ||
DOT → .
NOT → !
TYPE → int | float
LP → (
RP → )
LB → [
RB → ]
LC → {
RC → }
STRUCT → struct
RETURN → return
IF → if
ELSE → else
WHILE → while
```

High-level Definitions

```
Program → ExtDefList
ExtDefList → ExtDef ExtDefList
  | ε
ExtDef → Specifier ExtDecList SEMI
  | Specifier SEMI
  | Specifier FunDec CompSt
ExtDecList → VarDec
  | VarDec COMMA ExtDecList
```

[1] 你需要自行考虑如何用正则表达式表示整型数，你可以假设每个整型数不超过32bits位。

[2] 你需要自行考虑如何用正则表达式表示浮点数，你可以只考虑符合C语言规范的浮点常数（参见补充说明）。

[3] 你需要自行考虑如何用正则表达式表示标识符，你可以假设每个标识符长度不超过32个字符。

Specifiers

```
Specifier → TYPE
  | StructSpecifier
StructSpecifier → STRUCT OptTag LC DefList RC
  | STRUCT Tag
OptTag → ID
  | ε
Tag → ID
```

Declarators

```
VarDec → ID
  | VarDec LB INT RB
FunDec → ID LP VarList RP
  | ID LP RP
VarList → ParamDec COMMA VarList
  | ParamDec
ParamDec → Specifier VarDec
```

Statements

```
CompSt → LC DefList StmtList RC
StmtList → Stmt StmtList
  | ε
Stmt → Exp SEMI
  | CompSt
  | RETURN Exp SEMI
  | IF LP Exp RP Stmt
  | IF LP Exp RP Stmt ELSE Stmt
  | WHILE LP Exp RP Stmt
```

Local Definitions

```
DefList → Def DefList
  | ε
Def → Specifier DecList SEMI
DecList → Dec
  | Dec COMMA DecList
Dec → VarDec
  | VarDec ASSIGNOP Exp
```

Expressions

```
Exp → Exp ASSIGNOP Exp
  | Exp AND Exp
  | Exp OR Exp
  | Exp RELOP Exp
  | Exp PLUS Exp
  | Exp MINUS Exp
  | Exp STAR Exp
  | Exp DIV Exp
  | LP Exp RP
  | MINUS Exp
  | NOT Exp
  | ID LP Args RP
  | ID LP RP
  | Exp LB Exp RB
  | Exp DOT ID
  | ID
  | INT
```

```
  | FLOAT
Args → Exp COMMA Args
  | Exp
```

补充说明

Tokens

这一部分的产生式主要与词法有关:

1) 词法单元INT表示的是所有（无符号）整型常数。一个十进制整数由0~9十个数字组成，数字与数字中间没有如空格之类的分隔符。除“0”之外，十进制整数的首位数字不为0。例如，下面几个串都表示十进制整数：0、234、10000。为方便起见，你可以假设（或者只接受）输入的整数都在32bits位之内。

2) 整型常数还可以以八进制或十六进制的形式出现。八进制整数由0~7八个数字组成并以数字0开头，十六进制整数由0~9、A~F（或a~f）十六个数字组成并以0x或者0X开头。例如，0237（表示十进制的159）、0xFF32（表示十进制的65330）。

3) 词法单元FLOAT表示的是所有（无符号）浮点型常数。一个浮点数由一串数字与一个小数点组成，小数点的前后必须有数字出现。例如，下面几个串都是浮点数：0.7、12.43、9.00。为方便起见，你可以假设（或者只接受）输入的浮点数都符合IEEE754单精度标准（即都可以转换成C语言中的float类型）。

4) 浮点型常数还可以以指数形式（即科学记数法）表示。指数形式的浮点数必须包括基数、指数符号和指数三个部分，且三部分依次出现。基数部分由一串数字（0~9）和一个小数点组成，小数点可以出现在数字串的任何位置；指数符号为“E”或“e”；指数部分由可带“+”或“–”（也可不带）的一串数字（0~9）组成，“+”或“–”（如果有）必须出现在数字串之前。例如01.23E12（表示1.23×10^{12}）、43.e-4（表示43.0×10^{-4}）、.5E03（表示0.5×10^{3}）。

5) 词法单元ID表示的是除去保留字以外的所有标识符。标识符可以由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。为方便起见，你可以假设（或者只接受）标识符的长度小于32个字符。

6) 除了INT、FLOAT和ID这三个词法单元以外，其它产生式中箭头右边都表示具体的字符串。例如，产生式TYPE → int | float表示：输入文件中的字符串“int”和“float”都将被识别为词法单元TYPE。

High-level Definitions

这一部分的产生式包含了C--语言中所有的高层（全局变量以及函数定义）语法：

1) 语法单元Program是初始语法单元，表示整个程序。

2) 每个Program可以产生一个ExtDefList，这里的ExtDefList表示零个或多个ExtDef（像这种xxList表示零个或多个xx的定义下面还有不少，要习惯这种定义风格）。

3) 每个ExtDef表示一个全局变量、结构体或函数的定义。其中：

a) 产生式ExtDef → Specifier ExtDecList SEMI表示全局变量的定义，例如"int global1, global2;"。其中Specifier表示类型，ExtDecList表示零个或多个对一个变量的定义VarDec。

b) 产生式ExtDef → Specifier SEMI专门为结构体的定义而准备，例如"struct {…};"。这条产生式也会允许出现像"int;"这样没有意义的语句，但实际上在标准C语言中这样的语句也是合法的。所以这种情况不作为错误的语法（即不需要报错）。

c) 产生式ExtDef → Specifier FunDec CompSt表示函数的定义，其中Specifier是返回类型，FunDec是函数头，CompSt表示函数体。

Specifiers

这一部分的产生式主要与变量的类型有关：

1) Specifier是类型描述符，它有两种取值，一种是Specifier → TYPE，直接变成基本类型int或float，另一种是Specifier → StructSpecifier，变成结构体类型。

2) 对于结构体类型来说：

a) 产生式StructSpecifier → STRUCT OptTag LC DefList RC：这是定义结构体的基本格式，例如struct Complex { int real, image; }。其中OptTag可有可无，因此也可以这样写：struct { int real, image; }。

b) 产生式StructSpecifier → STRUCT Tag：如果之前已经定义过某个结构体，比如struct Complex {…}，那么之后可以直接使用该结构体来定义变量，例如struct Complex a, b;，而不需要重新定义这个结构体。

Declarators

这一部分的产生式主要与变量和函数的定义有关：

1) VarDec表示对一个变量的定义。该变量可以是一个标识符（例如int a中的a），也可以是一个标识符后面跟着若干对方括号括起来的数字（例如int a[10][2]中的a[10][2]，这种情况下a是一个数组）。

2) FunDec表示对一个函数头的定义。它包括一个表示函数名的标识符以及由一对圆括号括起来的一个形参列表，该列表由VarList表示（也可以为空）。VarList包括一个或多个ParamDec，其中每个ParamDec都是对一个形参的定义，该定义由类型描述符Specifier和变量定义VarDec组成。例如一个完整的函数头为：foo(int x, float y[10])。

Statements

这一部分的产生式主要与语句有关：

1) CompSt表示一个由一对花括号括起来的语句块。该语句块内部先是一系列的变量定义DefList，然后是一系列的语句StmtList。可以发现，对CompSt这样的定义，是不允许在程序的任意位置定义变量的，必须在每一个语句块的开头才可以定义。

2) StmtList就是零个或多个Stmt的组合。每个Stmt都表示一条语句，该语句可以是一个在末尾添了分号的表达式（Exp SEMI），可以是另一个语句块（CompSt），可以是一条返回语句（RETURN Exp SEMI），可以是一条if语句（IF LP Exp RP Stmt），可以是一条if-else语句（IF LP Exp RP Stmt ELSE Stmt），也可以是一条while语句（WHILE LP Exp RP Stmt）。

Local Definitions

这一部分的产生式主要与局部变量的定义有关：

1) DefList这个语法单元前面曾出现在CompSt以及StructSpecifier产生式的右边，它就是一串像int a; float b, c; int d[10];这样的变量定义。一个DefList可以由零个或者多个Def组成。

2) 每个Def就是一条变量定义，它包括一个类型描述符Specifier以及一个DecList，例如int a, b, c;。由于DecList中的每个Dec又可以变成VarDec ASSIGNOP Exp，这允许我们对局部变量在定义时进行初始化，例如int a = 5;。

Expressions

这一部分的产生式主要与表达式有关：

1) 表达式可以演化出的形式多种多样，但总体上看不外乎下面几种：

a) 包含二元运算符的表达式：赋值表达式（Exp ASSIGNOP Exp）、逻辑与（Exp AND Exp）、逻辑或（Exp OR Exp）、关系表达式（Exp RELOP Exp）以及四则运算表达式（Exp PLUS Exp等）。

b) 包含一元运算符的表达式：括号表达式（LP Exp RP）、取负（MINUS Exp）以及逻辑非（NOT Exp）。

c) 不包含运算符但又比较特殊的表达式：函数调用表达式（带参数的ID LP Args RP以及不带参数的ID LP RP）、数组访问表达式（Exp LB Exp RB）以及结构体访问表达式（Exp DOT ID）。

d) 最基本的表达式：整型常数（INT）、浮点型常数（FLOAT）以及普通变量（ID）。

2) 语法单元Args表示实参列表，每个实参都可以变成一个表达式Exp。

3) 由于表达式中可以包含各种各样的运算符，为了消除潜在的二义性问题，我们需要给出这些运算符的**优先级（precedence）**以及**结合性（associativity）**，如表1所示。

Comments

表1. C--语言中运算符的优先级和结合性。

优先级[1]	运算符	结合性	描述
1	(,)	左结合	括号或函数调用。
	[,]		数组访问。
	.		结构体访问。
2	–	右结合	取负。
	!		逻辑非。
3	*	左结合	乘。
	/		除。
4	+		加。
	–		减。
5[2]	<		小于。
	<=		小于或等于。
	>		大于。
	>=		大于或等于。
	==		等于。
	!=		不等于。
6	&&		逻辑与。
7	||		逻辑或。
8	=	右结合	赋值。

C--源代码可以使用两种风格的注释：一种是使用双斜线“//”进行单行注释，在这种情况下，该行在“//”符号之后的所有字符都将作为注释内容而直接被词法分析程序丢弃掉；另一种是使用“/*”以及“*/”进行多行注释，在这种情况下，在“/*”与之后最先遇到的“*/”之间的所有字符都被视作注释内容。需要注意的是，“/*”与“*/”是不允许嵌套的：即在任意一对“/*”和“*/”之间不能再包含成对的“/*”和“*/”，否则编译器需要进行报错。

[1] 数值越小表示代表优先级越高。

[2] 在标准C语言中，“>”、“<”、“>=”和“<=”四个关系运算符的优先级要比“==”和“!=”两个运算符的优先级高；在这里，我们为了方便处理而将它们的优先级统一了。

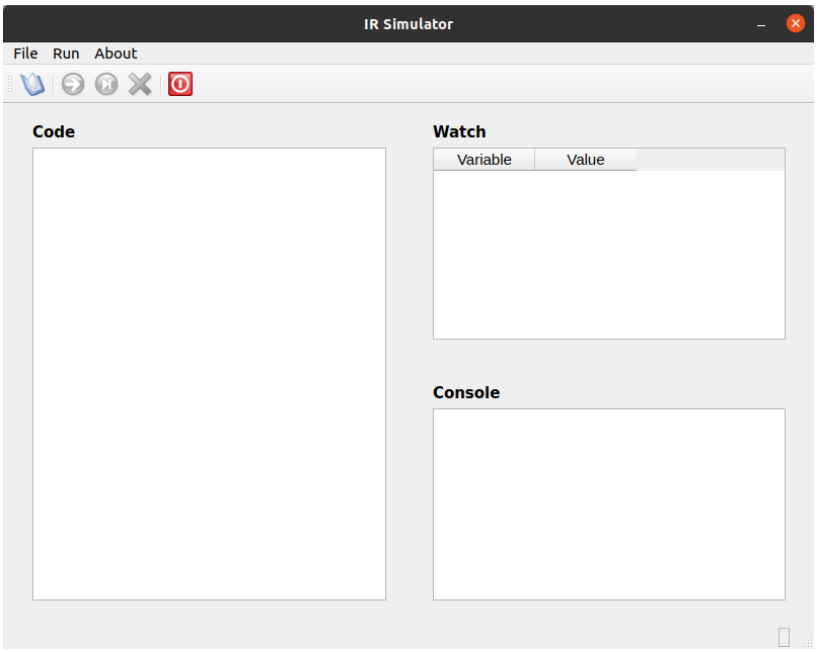

图1. 虚拟机小程序IR Simulator的界面。

附录B：虚拟机小程序使用说明

我们提供的虚拟机小程序叫做IR Simulator，它本质上就是一个中间代码解释器。这个程序使用Python写成，图形界面部分则借助了跨平台的诺基亚Qt库。由于实验环境为Linux，因此该程序只在Linux下发布。注意，运行虚拟机小程序之前要确保本机上装有Qt运行环境。你可以在Linux终端下使用“pip install sip PyQt5 PyQt5-tools”命令以获取Qt运行环境（也可参考附录C中的离线安装方式），然后通过我们课程网站下载虚拟机小程序（请参考附录C中的资源下载和安装介绍）。如果出现了qt.qpa.plugin:xcb插件问题，可以使用“sudo apt install libxcb-xinerama0”命令下载依赖库。

IR Simulator的运行界面如图1所示。整个界面分为三个部分：左侧的**代码区**、右上方的**监视区**和右下方的**控制台区**。代码区显示已经载入的中间代码，监视区显示中间代码中当前执行函数的参数与局部变量的值，控制台区供WRITE函数进行输出用。我们可以点击上方工具栏中的第一个按钮或通过菜单选择File → Open来打开一个保存有中间代码的文本文件（注意该文件的后缀名必须是ir）。如果中间代码中包含语法错误，则IR Simulator会弹出对话框提示出错位置并拒绝继续载入代码；如果中间代码中没有错误，那么你将看到类似于图2所示的内容。

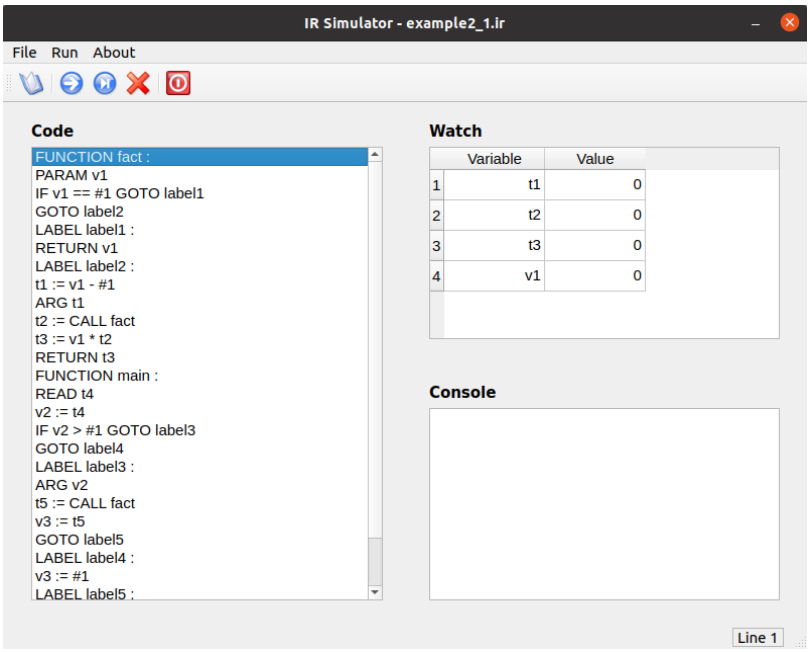

图2. IR Simulator载入中间代码成功的界面。

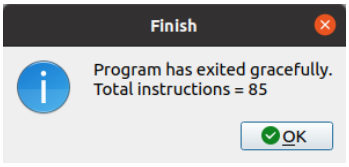

图3. IR Simulator运行中间代码成功的提示。

此时中间代码已被完全载入到左侧的代码区，因为中间代码尚未执行，所以右侧监视区的显示为空。此时你可以单击工具栏上的第二个按钮或通过菜单选择Run → Run（快捷键为F5）直接运行该中间代码，或者单击第三个按钮或通过菜单选择Run → Step（快捷键为F8）来对该中间代码进行单步执行。单击第四个按钮或通过菜单选择Run → Stop可以停止中间代码的运行并重新初始化。如果中间代码的运行出现错误，那么IR Simulator会弹出对话框以提示出错的行号以及原因（一般的出错都是因为访问了一个不存在的内存地址，或者在某个函数中缺少RETURN语句等）；如果一切正常，你将会看到如图3所示的对话框。

这提示我们中间代码的运行已经正常结束，并且本次运行共执行了85条指令。由于实验三会考察优化中间代码的效果，如果你想得到更高的评价，就需要设法使Total instructions后面的数字尽可能小。

另外，关于IR Simulator有几点需要注意：

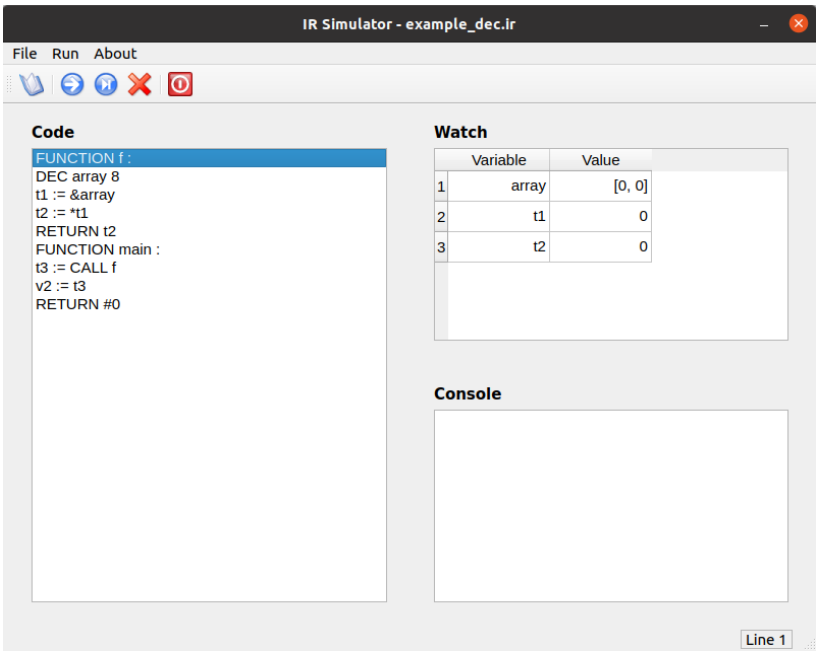

图4. DEC定义的变量与普通变量之间的显示区别。

1) 在运行之前请确保你的中间代码不会陷入死循环或无穷递归中。IR Simulator不会对这类情况进行判断，因此一旦当它执行了包含死循环或无穷递归的中间代码，你就需要将IR Simulator强制退出了。

2) 互相对应的ARG语句和PARAM语句数量一定要相等，否则可能出现无法预知的错误。

3) 由于IR Simulator实现上的原因，单步执行时它并不会在GOTO、IF、RETURN等与跳转相关的语句上停留，而是会将控制直接转移到跳转目标那里，这是正常的情况。

4) 在中间代码的执行过程中，右侧监视区始终显示当前正在执行函数的中间代码中的变量及其值。如图4中所示，当执行到函数“f”的第一条代码时，右侧监视区中会显示此函数对应的中间代码中使用的变量，且变量的初始值默认为0。

5) 使用DEC语句定义的变量在监视区中的显示与普通的变量的区别如图4所示。注意监视区中变量array的值：与其它变量不同，array中的内容被一对中括号“[”和“]”包裹了起来。有时候，因为DEC出来的变量内容较多，Value一栏可能无法显示完全，你可以用鼠标调整Value栏的宽度使其所有内容都能被显示出来。

6) 所有函数的参数与局部变量都只在运行到该函数之后才会去为其分配存储空间并在监视区显示。存储空间采用由低地址到高地址的栈式分配方式，递归调用的局部变量将不会影响到上层函数的相应变量的值。如果想要修改上层函数中变量的值，则需要向被调用的函数传递

该变量的地址作为参数，也就是我们常说的引用调用。不过，监视区中显示的变量总是当前这一层变量的值，只要不退回到上一层，我们就无法看到上层函数局部变量的值是多少。

附录C：资源下载和安装介绍

在本附录中，我们介绍编译实验环境的搭设。有两种方式：一是在连接Internet的条件下进行在线软件安装，二是利用随书附赠的光盘进行离线软件安装。

在线软件安装方式

在线软件安装的步骤如下：

1) 在Ubuntu官方网站上下载Ubuntu 20.04操作系统并进行安装，其下载地址为：https://releases.ubuntu.com/20.04/ubuntu-20.04.5-desktop-amd64.iso.torrent?_ga=2.213851344.515239749.1664973589-379588267.1664973589，文件名为：ubuntu-20.04.5-desktop-amd64.iso。

2) 在Ubuntu终端使用命令“sudo apt-get install flex”以安装flex软件。安装完成后，可使用命令“flex --version”来查看flex的版本号（目前版本为“flex 2.6.4”）。

3) 在Ubuntu终端使用命令“sudo apt-get install bison”以安装bison软件。安装完成后，可使用命令“bison --version”来查看bison的版本号（目前版本为“bison 3.5.1”）。

4) 在Ubuntu终端使用命令“pip install sip PyQt5 PyQt5-tools”以安装Qt运行环境。

5) 从课程群下载我们提供的虚拟机小程序（用于中间代码执行）。将下载得到的压缩包中的三个文件解压到同一用户目录下，并在该目录下执行“python3 irsim.pyc”即可启动虚拟机小程序。

6) 从SPIM Simulator的官方网站上下载QtSPIM软件（用于目标代码执行），其下载地址为：http://pages.cs.wisc.edu/~larus/spim.html，文件名为：qtspim_9.1.9_linux32.deb。双击该文件以进行安装，在安装过程中会弹出“The package is of bad quality”的提示框，这时选择“Ignore and install”可继续安装。

如果以上软件的版本号与我们说明的不完全一致，在大多数情况下并不会影响编译实验的进行。如果遇到特殊情况或需要完全一致，也可参照下面的离线软件安装。