运用于 Minisys-1A 的 Mini C 的有关规定

一、数据类型、运算符和表达式

I 数据类型:

a. 目前支持的数包括

- 32 位有符号整数型-int
- 32 位无符号整型数-unsigned
- 16 位有符号数-short
- 16 位无符号数-unsigned short
- 8 位有符号数-char
- 8 位无符号-unsigned char
- 32 位整型指针类型的数据-int *

b. 常量支持十六进制表示 0xff00 与十进制表示方式，最大支持 32bit，超出部分将被截取

c. 标识符只能为字母数字串，不能有下划线和其他符号。

d. 函数名不能为'label+数字'这种形式。

II 运算符号:

e. 算数运算

运算符	类型	含义
+	双目	加
-	双目	减
*	双目	乘
/	双目	除
%	双目	取余
+	单目	正
-	单目	负

算数运算中，单目都是右结合，双目是左结合，单目优先级高于双目，乘除高于加减

f. 关系运算:

运算符	含义
<	小于
>	大于

<=	小于等于
>=	大于等于
==	等于
!=	不等

g. 逻辑运算:

运算符	含义
&&	与
||	或
!	非

h. 位运算:

运算符	含义	备注
&	按位与	
|	按位或	
^	按位异或	
~	按位取反	单目，右结合

i. 指针类型数的运算符号

运算符	含义	备注
*	取指针数据的内容	一般只用于对 I/O 端口进行访问

j. 优先级声明:

算数运算>关系运算>逻辑运算,括号优先级最高.

III 表达式.

k. 赋值表达式与变量声明:必须遵守先定义，再赋值的规则.支持 int a,b,c;声明方式。

Eg. int a;

a=10;是合法的。

但:int a=10;是非法的。

l. 函数内部使用自定义变量时，必须在函数开始处，定义好全部要使用的变量，不允许在块内和其他地方再定义变量.

Eg: void fun(void)

```
{
int a;
a=1;
}
```

合法

Eg: void fun(void)

```
{
int a;
```

```
a=1;
int b;
}
```

不合法，不允许在变量未声明完之前使用任何变量.具体的讨论在函数部分.

二、一个简单的 Mini C 程序

例:

```
void delay(void)
{
        int c;c=30000;
          while(c>0){ c=c-1;}
}
void main(void)
{
    int key;
    int *LED;
    key=0;
    LED = 0xfffffc60
     while(1)
        {
                  key=key+1;
                  *LED=key;  //输出到 LED 灯
                    if(key>10) key=0;
                    delay();
        }
}
```

这是一个简单的在 MiniSys 系统实验板的 LED 数码管上循环显示 1-A 的程序，风格和 ANSI C 的风格基本一样。

三、程序控制语句

I.循环控制

支持 while(expr)循环，while 循环内部可以使用 continue,break 控制循环，也可以嵌套 while 循环和 if-else 控制语句。

while 循环控制条件可以为表达式 expr 不为 0，可以是变量，常数，算数表达式和关系，逻辑表达式.

eg:

```
while(c>0){ c=c-1;}
```

和

```
while(1)
```

```
{
            key=key+1;
          $0xfffffc00=key;
          if(key>10) key=0;
          delay();
}
```

II. 条件控制:

支持 if,if-else 结构,可以嵌套 if,if-else 结构。条件控制表达式和 while 相同.

eg:

```
if(n<=0) return 0;
if(n==1||n==2) return 1;
else return fib(n-1)+fib(n-2);
```

四、函数

I. 函数的定义和使用.

a. 函数的返回类型只能是 void 或 int 型，函数可以有参数，也可以没参数，但函数没参数时，声明时，参数列表需要用 void 填充。可以先只有函数体的申明，再在后面完成函数体。

b. 当一维数组作为参数申明时，数组必须带常数下标，常数下标可以为任何正值，但不可以没有。

c. return 执行函数返回。

d. 支持定义递归函数，支持函数嵌套

e. 调用不含参数的函数时，参数表不能填写 void.

例 1. 递归函数

```
int fib(int n)
{
          if(n<=0) return 0;
            if(n==1||n==2) return 1;
            else return fib(n-1)+fib(n-2);
}
```

例 2. 一维数组作为参数

```
void set(int arr[10],int n,iny val)
{
          arr[n]=val;
}
```

调用这个函数:

```
int b[10];
set(b,5,200);
```

例 3. 不带参数的函数

```
void delay(void)
{
int c;c=3000000;
while(c>0){ c=c-1;}
}
```
调用: delay();

f. 中断函数，中断函数不可以有返回值与参数,必须声明为下面的形式

```
void interuptServer0(void);
void interuptServer1(void);
```
用于响应 int0 和 int1 中断。

例子:

```
int a;
void interuptServer0(void)
{
          int b;
          int c;
          b = a&0xf000;
          c = a + 1;
          c = c & 0x0fff;
          a = ~(~b&~c);
}
void interuptServer1(void)
{
          int b;
          int c;
          b=a&0x0fff;
          c = $0xfffffc10;
          c = c<<12;
          a = ~(~b&~c);
}
void main(void)
{
          $0xfffffc20 = 2;
          $0xff22 = 0x000fffff;
          a = 0;
          while(1)
        {
                  $0xff00 = a;
    }
}
```
功能是,数码管前三位是秒表，后面一位显示键盘值.通过键盘中断和计数器秒中断驱动.

五、数组的使用

I.数组定义

类型 数组名 [正整数] ;只支持一维数组.
eg: int a[10];
同样只支持一个一个地声明，不支持 int a[10],b[10];

II.数组的访问

通过: 数组名[下标变量]方式进行访问;
eg:
int a[10];
int b;
a[5]=100;
a[1]=5;
b=a[1];
b=a[b];
最终:b=100;

III.向函数传递一维数组

int g[10];
set(g,3,77);
本质上，传给函数的是数组的首地址，相当于指针，用户可以通过数组传递大量的数据给函数，也可以从函数获得大量的返回信息。由于不进行下标检查，请用户谨慎使用数组。

六、端口操作

使用 32 位整型指针操作端口。
以下是一个例子

```
int main() {
  int * RSWITCHES;
  int *WLED;
  RSWITCHES = 0xfffffc70;
  WLED = 0xfffffc60;
  unsigned int switches;

  while (1) {
    switches = *RSWITCHES;
    *WLEDR = switches;
  }
  return 0;
}
```

地址空间说明:

0000H-07F4H:用户程序内存空间
1FFF8H:中断 0 向量入口
1FFFCH:中断 1 向量入口
I/O 端口地址如下：
Minisys-1A 的地址端口定义：(数据都是 16 位)

数码管	0FFFFFC00H
4×4 键盘	0FFFFFC10H
定时器	0FFFFFC20H
PWM	0FFFFFC30H
UART	0FFFFFC40H
看门狗	0FFFFFC50H
Led	0FFFFFC60H
拨码开关	0FFFFFC70H
VGA	0FFFFFC80H
三轴加速器	0FFFFFC90H
101 键盘	0FFFFFCA0H
鼠标	0FFFFFCB0H
GPI	0FFFFFCC0H

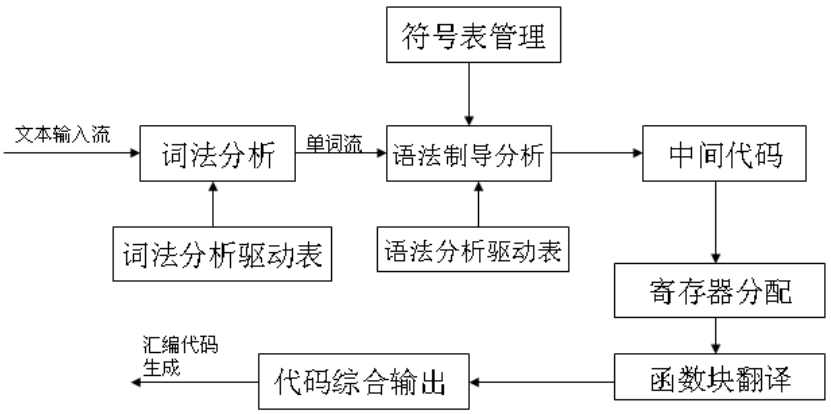
具体定义请看课件

请大家根据以上规定设计相关的词法规则和语法规则。

以下是一个范例，供大家参考。

➢ 总体设计

MiniC 编译器主要由以下几个部分组成,词法分析器,语法制导分析筐架（含中间代码生成）,寄存器分配模块，函数块翻译模块，代码综合输出模块。结构示意图如下:



词法分析器在词法分析表的驱动下，从输入流中识别符号，为语法分析提供单词流。语法分析器采用了 LALR(1)分析法，并采用语法指导的方式进行中间代码的生成，符号表管理模块为语法分析过程提供单词登记，查询的功能，并参与变量的空间分配计算，为后续的代码生成，提供地址信息。寄存器分配部分负责对翻译过程中产生的临时变量进行寄存器关联，采用的策略是简单统计各个临时变量的使用频率，为使用最频繁的前 10 个临时变量分配寄存器，对超出部分的寄存器，为其分配内存空间。函数块的翻译负责将中间代码生成部分产生的各个函数的中间代码翻译为汇编代码，并加上相应的现场保护，现场恢复的代码，完成子程序的翻译。最后的代码综合输出主要作用是，将各个子程序进行总装，加上中断入口，程序初始化代码，产生完整的一个汇编代码，完成翻译。产生的汇编代码只要通过 Minisys 的汇编器翻译，就可以生成最终的机器代码，进入 CPU 运行。

- **词法规则(Lex 描述)** （不完全符合上述规定，大家自行修改）

```
Digit            ([0-9])|([1-9][0-9]*)
letter           [a-zA-Z]
id               {letter}({letter}|{digit})*
num              {digit}[1-9]*
hex              0(x|X)({letter}|[0-9])*
%%
"void"          {return VOID;}
"continue"      {return CONTINUE;}
"if"           {return IF; }
"while"          {return WHILE;}
"else"         {return ELSE;}
"break"         {return BREAK;}
"int"          {return INT;}
"return"        {return RETURN;}
"\|\|"          {return OR;}
"&&"            {return AND;}
{id}            {yyval.IDENT_v.ID_NAME=yytext;return IDENT;}
 {hex}
                 {yyval.int_literal_v.int_val=ConvertHexToint(yyt
                 ext);return HEXNUM;}
    {num}          {yyval.int_literal_v.int_val=atoi(yytext);return
DECNUM;}
    "\<="          {return LE;}
    "\>="          {return GE;}
    "\=\="         {return EQ;}
    "\!\="         {return NE;}
    "\>"           {return '>';}
    "\<"           {return '<';}
```

```
"\,"            {return ','; }
"\;"           {return ';'; }
"\{"           {return '{'; }
"\}"           {return '}'; }
"\%"           {return '%'; }
"\*"           {return  '*';}
"\+"           {return '+'; }
"\-"           {return '-'; }
"/"            {return '/'; }
"\="             {return '='; }
"\("           {return '('; }
"\)"           {return ')'; }
"~"              {return '~';}
"&"              {return '&';}
"^"              {return '^';}
"\["             {return '[';}
"\]"             {return ']';}
"\<\<"           {return LSHIFT;}
"\>\>"           {return RSHIFT;}
"\|"             {return '|';}
\t|\             { }
\n|\r\n          {Lineno++;yylineno++;}
"$"              {return '$';}
%%
```

- **语法规则(Yacc 描述)** （不完全符合上述规定，大家自行修改）

```
%token IDENT VOID INT WHILE IF ELSE RETURN EQ NE LE GE AND OR DECNUM
CONTINUE BREAK HEXNUM LSHIFT RSHIFT
%left OR
%left AND
%left EQ NE LE GE '<' '>'  /*关系运算*/
%left '+' '-'
%left '|'
%left '&' '^'
%left '*' '/' '%'  /*算术运算*/
%right LSHIFT RSHIFT
%right '!'
%right '~'
%nonassoc UMINUS
%nonassoc MPR
%start program
%%
```

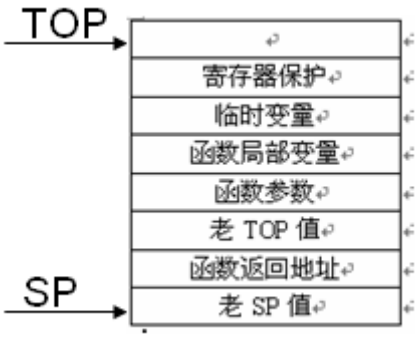
```
program  : decl_list  ; /*程序由变量描述或函数描述组成（decl）*/
decl_list  : decl_list decl | decl ;
decl  : var_decl | fun_decl ;
var_decl  : type_spec IDENT ';' | type_spec IDENT '[' int_literal ']' ';' ; /*变量包括简单变量和一维数组变量*/
type_spec  : VOID | INT ; /*函数返回值类型或变量类型包括整型或VOID*/
fun_decl  : type_spec FUNCTION_IDENT '(' params ')' compound_stmt
      | type_spec FUNCTION_IDENT '(' params ')' ';' ; //要考虑设置全局函数信息为假，和函数表为申明
FUNCTION_IDENT : IDENT {cout<<"function ident"<<endl;} ; /*建立全局函数名变量*/
params  :  param_list | VOID ; /*函数参数个数可为0或多个*/
param_list  : param_list ',' param | param ;
param  :  type_spec IDENT | type_spec IDENT '[' int_literal ']' ;
stmt_list  : stmt_list stmt | ;
stmt  :  expr_stmt | block_stmt | if_stmt | while_stmt |return_stmt | continue_stmt | break_stmt  ;
expr_stmt  : IDENT '=' expr ';' | IDENT '[' expr ']' '=' expr ';' | '$' expr '=' expr ';' | IDENT '(' args ')' ';' ; /*赋值语句*/
while_stmt : WHILE_IDENT '(' expr ')' stmt ; /*WHILE语句*/
WHILE_IDENT : WHILE {cout<<"while ident"<<endl;} ;/*建立入口出口信息全局变量*/
block_stmt : '{' stmt_list '}'  ; /*语句块*/
compound_stmt : '{' local_decls stmt_list '}'  ; /*函数内部描述，包括局部变量和语句描述*/
local_decls  : local_decls local_decl | ; /*函数内部变量描述*/
local_decl  : type_spec IDENT ';' | type_spec IDENT '[' int_literal ']' ';' ;
if_stmt  :  IF '(' expr ')' stmt %prec UMINUS | IF '(' expr ')'  stmt ELSE stmt %prec MPR ;
return_stmt  : RETURN ';' | RETURN expr ';' ;
expr  :  expr OR expr  /*逻辑或表达式，运算符为'||'*/
  | expr EQ expr | expr NE expr /*关系表达式*/
  | expr LE expr | expr '<' expr | expr GE expr | expr '>' expr /*关系表达式*/
  | expr AND expr /*逻辑与表达式，运算符为'&&'**/
  | expr '+' expr | expr '-' expr /*算术表达式*/
  | expr '*' expr | expr '/' expr | expr '%' expr  /*算术表达式*/
  | '!' expr %prec UMINUS | '-' expr %prec UMINUS | '+' expr %prec UMINUS | '$' expr %prec UMINUS  /*$ expr为取端口地址为expr值的端口值*/
  | '(' expr ')'
```

```
    | IDENT | IDENT '[' expr ']' | IDENT '(' args ')'   /* IDENT ( args ) 为函数调用*/
    | int_literal   /*数值常量*/
    | expr '&' expr /*按位与*/
    | expr '^' expr /*按位异或*/
    | '~' expr   /*按位取反*/
    | expr LSHIFT expr /*逻辑左移*/
    | expr RSHIFT expr /*逻辑右移*/
    | expr '|' expr /*按位或*/
     ;
int_literal  :  DECNUM
             |  HEXNUM ; /*数值常量是十进制整数*/
arg_list   :  arg_list ',' expr | expr ;
args   :   arg_list | ;
continue_stmt : CONTINUE ';' ;
break_stmt : BREAK ';' ;
%%
```

注:本程序的词法和语法分析器由我们自己设计的 SeuLex+SeuYacc 工具生成，详细细节请参考 SeuLex 和 SeuYacc 的设计文档。

➢ 运行时的内存布局

MiniC 编译器产生的程序,运行时的活动记录基本类似于 ANSI C 的结构，考虑到 MiniC 对 C 语言进行了裁剪，对活动记录进行了适应性修改，活动记录采用了向上生长的方式，具体结构如下:

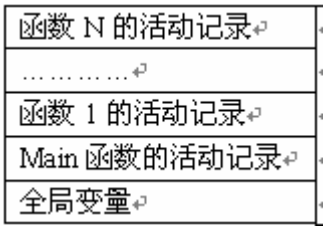


SP 是指向当前活动记录底部的指针，TOP 是指向当前活动记录顶部的第一个空单元的指针。

当函数调用发生时，产生一个新的活动记录，当一个函数返回时，它的活动记录也被释放，从而实现了一个栈式动态分配。

- 活动记录的创建。当一个函数调用发生时，向 8(TOP),0(TOP)保存老 SP 和老 TOP,保护现场,初始化新的 SP 为 SP=TOP,新 TOP 初始化为老 TOP+新活动记录的大小。
- 活动记录的销毁和释放。当一个函数返回时,先完成现场的恢复，恢复

TOP=8(SP),恢复 SP=0(SP)。



一般程序的内存布局

➢ 中间代码到汇编代码翻译的技巧

中间代码在完成临时变量的寄存器分配后，一共有三种变量形式，常数，t 类寄存器(10 个),内存数，为了简化翻译过程,以四个 s 寄存器为基础，加入 MOV 中间指令，负责在三种变量类型之间进行转换，从而省略了其他中间指令翻译的代码量，实现了模块的最大化重用。MOV 中间指令的思想很简单，借鉴了 X86 指令集的 MOV,同时考虑到 MIPS 指令的规整性,以及 Mini C 的特点，将数据转换传送过程交给 MOV 处理，解决了代码重复问题。

➢ 错误检测

提供一定的语法错误检测能力和丰富的语义错误检测功能。对于非法的符号，词法分析器有一定的能力跳过,从而使分析过程能够继续下去。语义检测功能中，提供了很好的类型匹配和参数匹配检测，像 ANSI C 一样，不提供对数组下标的检查。下面是一个简单的例子:

```
int fib(int n);
void main(void)
{ fib(1,2); }
```

编译器会检测出 fib 函数的参数数目不匹配，并给出提示。具体的细节，参阅使用说明。