# 附录A　编译器设计方案

**本章要点**

- C－惯用的词法
- C－的语法和语义
- C－的程序例子
- C－语言的Tiny Machine运行时环境
- 使用C－和TM的编程设计

这里定义了一个编程语言称作**C－Minus** (或简称为C－)，这是一种适合编译器设计方案的语言，它比TINY语言更复杂，包括函数和数组。本质上它是C的一个子集，但省去了一些重要的部分，因此得名。这个附录由5小节组成。首先，我们列出了语言惯用的词法，包括语言标记的描述。其次，给出了每个语言构造的BNF描述，同时还有相关语义的英语描述。在A.3节，给出了C－的两个示例程序。再者，描述了C－的一个Tiny Machine运行时环境。最后一节描述了一些使用C－和TM的编程设计方案，适合于一个编译器教程。

## A.1　C－惯用的词法

1. 下面是语言的关键字：

```
else if int return void while
```

所有的关键字都是保留字，并且必须是小写。

2. 下面是专用符号：

```
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
```

3. 其他标记是***ID***和***NUM***，通过下列正则表达式定义：

```
ID = letter letter*
NUM = digit digit*
letter = a|..|z|A|..|Z
digit = 0|..|9
```

小写和大写字母是有区别的。

4. 空格由空白、换行符和制表符组成。空格通常被忽略，除了它必须分开 ***ID***、***NUM***关键字。
5. 注释用通常的C语言符号**/*...*/**围起来。注释可以放在任何空白出现的位置 (即注释不能放在标记内)上，且可以超过一行。注释不能嵌套。

## A.2　C－的语法和语义

C－的BNF语法如下：

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*

4. *var-declaration* → *type-specifier* **ID;** | *type-specifier* **ID [ NUM ];**
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier* **ID (** *params* **)** | *compound-stmt*
7. *params* → *params-list* | **void**
8. *param-list* → *param-list* **,** *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID [ ]**
10. *compound-stmt* → **{** *local-declarations statement-list* **}**
11. *local-declarations* → *local-declarations var-declaration* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt*
    | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* **;** | **;**
15. *selection-stmt* → **if (** *expression* **)** *statement*
    | **if (** *expression* **)** *statement* **else** *statement*
16. *iteration -stmt* → ***while*** **(** *expression* **)** *statement*
17. *return -stmt* → **return ;**| **return** *expression***;**
18. *expression* → *var* **=** *expression* | *simple-expression*
19. *var* → **ID** | **ID [** *expression* **]**
20. *simple-expression* → *additive-expression relop additive-expression*
    | *additive -expression*
21. *relop* →**<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* →**+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* →***** | **/**
26. *factor* → **(** *expression* **)** | *var* | *call* | **NUM**
27. *call* → **ID (** *args* **)**
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* **,** *expression* | *expression*

对以上每条文法规则，给出了相关语义的简短解释。

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*

程序由声明的列表(或序列)组成，声明可以是函数或变量声明，顺序是任意的。至少必须有一个声明。接下来是语义限制(这些在C中不会出现)。所有的变量和函数在使用前必须声明(这避免了向后backpatching引用)。程序中最后的声明必须是一个函数声明，名字为**main**。注意，C－缺乏原型，因此声明和定义之间没有区别(像C一样)。

4. *var-declaration* → *type-specifier* **ID ;** | *type-specifier* **ID [** **NUM** **];**
5. *type-specifier* → **int** | **void**

变量声明或者声明了简单的整数类型变量，或者是基类型为整数的数组变量，索引范围从 0到**NUM** - 1。注意，在C－中仅有的基本类型是整型和空类型。在一个变量声明中，只能使用类型

指示符**int**。**void**用于函数声明(参见下面)。也要注意，每个声明只能声明一个变量。

6. *fun-declaration* → *type-specifier* ***ID*** **(** *params* **)***compound-stmt*

7. *params* → *param-list* | **void**

8. *param-list* → *param-list* **,** *param* | *param*

9. *param* → *type-specifier* ***ID*** | *type-specifier* ***ID*** **[ ]**

函数声明由返回类型指示符、标识符以及在圆括号内的用逗号分开的参数列表组成，后面跟着一个复合语句，是函数的代码。如果函数的返回类型是 **void**，那么函数不返回任何值(即是一个过程)。函数的参数可以是**void** (即没有参数)，或者一列描述函数的参数。参数后面跟着方括号是数组参数，其大小是可变的。简单的整型参数由值传递。数组参数由引用来传递(也就是指针)，在调用时必须通过数组变量来匹配。注意，类型“函数”没有参数。一个函数参数的作用域等于函数声明的复合语句，函数的每次请求都有一个独立的参数集。函数可以是递归的(对于使用声明允许的范围)。

10. *compound-stmt* → **{** *local-declarations statement-list* **}**

复合语句由用花括号围起来的一组声明和语句组成。复合语句通过用给定的顺序执行语句序列来执行。局部声明的作用域等于复合语句的语句列表，并代替任何全局声明。

11. *local-declarations* → *local-declarations var-declaration* | *empty*

12. *statement-list* → *statement-list statement* | *empty*

注意声明和语句列表都可以是空的(非终结符*empty*表示空字符串，有时写作ε。)

13. *statement* → *expression-stmt*
| *compound-stmt*
| *selection-stmt*
| *iteration-stmt*
| *return-stmt*

14. *expression-stmt* → *expression* **;** | **;**

表达式语句有一个可选的且后面跟着分号的表达式。这样的表达式通常求出它们一方的结果。因此，这个语句用于赋值和函数调用。

15. *selection-stmt* → **if (***expression***)** *statement*
| **if (***expression***)** *statement* **else** *statement*

if语句有通常的语义：表达式进行计算；非0值引起第一条语句的执行；0值引起第二条语句的执行，如果它存在的话。这个规则导致了典型的悬挂 else二义性，可以用一种标准的方法解决：else部分通常作为当前if的一个子结构立即分析(“最近嵌套”非二义性规则)。

16. *iteration-stmt* → **while (***expression***)** *statement*

while语句是C－中唯一的重复语句。它重复执行表达式，并且如果表达式的求值为非 0，则执行语句，当表达式的值为0时结束。

17. *return -stmt* → **return ;**| **return** *expression***;**

返回语句可以返回一个值也可无值返回。函数没有说明为 **void**就必须返回一个值。函数声明为**void**就没有返回值。return引起控制返回调用者(如果它在**main**中，则程序结束)。

18. *expression* → *var* = *expression* | *simple-expression*

19. *var* → ***ID*** | ***ID*** **[***expression***]**

表达式是一个变量引用，后面跟着赋值符号 (等号)和一个表达式，或者就是一个简单的表达式。赋值有通常的存储语义：找到由 *var*表示的变量的地址，然后由赋值符右边的子表达式

进行求值，子表达式的值存储到给定的地址。这个值也作为整个表达式的值返回。 *var*是简单的(整型)变量或下标数组变量。负的下标将引起程序停止 (与C不同)。然而，不进行下标越界检查。

var表示C－比C的进一步限制。在C中赋值的目标必须是左值(**l-value**)，左值是可以由许多操作获得的地址。在C－中唯一的左值是由*var*语法给定的，因此这个种类按照句法进行检查，代替像C中那样的类型检查。故在C－中指针运算是禁止的。

20. *simple-expression* → *additive-expression relop additive-expression*
| *additive -expression*

21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**

简单表达式由无结合的关系操作符组成 (即无括号的表达式仅有一个关系操作符 )。简单表达式在它不包含关系操作符时，其值是加法表达式的值，或者如果关系算式求值为 ture，其值为1，求值为false时值为0。

22. *additive-expression* → *additive-expression addop term* | *term*

23. *addop* → **+** | **-**

24. *term* → *term mulop factor* | *factor*

25. *mulop* → ***** | **/**

加法表达式和项表示了算术操作符的结合性和优先级。符号表示整数除；即任何余数都被截去。

26. *factor* → **(***expression***)** | *var* | *call* | ***NUM***

因子是围在括号内的表达式；或一个变量，求出其变量的值；或者一个函数调用，求出函数的返回值；或者一个***NUM***，其值由扫描器计算。数组变量必须是下标变量，除非表达式由单个***ID*** 组成，并且以数组为参数在函数调用中使用 (如下所示)。

27. *call* → ***ID*** **(** *args* **)**

28. *args* → *arg-list* | *empty*

29. *arg-list* → *arg-list* **,** *expression* | *expression*

函数调用的组成是一个***ID*** (函数名)，后面是用括号围起来的参数。参数或者为空，或者由逗号分割的表达式列表组成，表示在一次调用期间分配的参数的值。函数在调用之前必须声明，声明中参数的数目必须等于调用中参数的数目。函数声明中的数组参数必须和一个表达式匹配，这个表达式由一个标识符组成表示一个数组变量。

最后，上面的规则没有给出输入和输出语句。在 C－的定义中必须包含这样的函数，因为与C不同，C－没有独立的编译和链接工具；因此，考虑两个在全局环境中预定义的函数，好像它们已进行了声明：

```
int input(void) {...}
void output(int x) {...}
```

**input**函数没有参数，从标准输入设备 (通常是键盘)返回一个整数值。**output**函数接受一个整型参数，其值和一个换行符一起打印到标准输出设备 (通常是屏幕)。

## A.3 C－的程序例子

下面的程序输入两个整数，计算并打印出它们的最大公因子。

```
/* A program to perform Euclid's
   Algorithm to compute gcd. */
```

```
int gcd (int u, int v)
{ if (v == 0) return u ;
  else return gcd(v,u-u/v*v);
  /* u-u/v*v == u mod v */
}

void main(void)
{ int x; int y;
  x = input(); y = input();
  output(gcd(x,y));
}
```

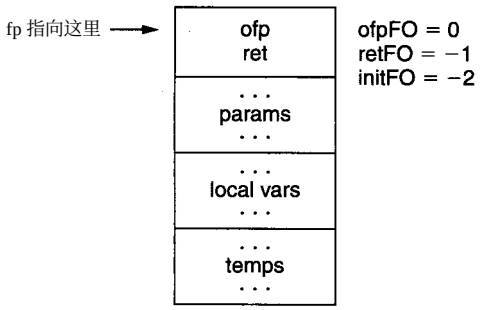
下面的程序输入10个整数的列表，对它们进行选择排序，然后再输出：

```
/* A program to perform selection sort on a 10
   element array. */

int x[10];

int minloc ( int a[], int low, int high )
{ int i; int x; int k;
  k = low;
  x = a[low];
  i = low + 1;
  while (i < high)
    { if (a[i] < x)
        {  x = a[i];
          k = i; }
      i = i + 1;
    }
  return k;
}

void sort ( int a[], int low, int high )
{ int i; int k;
  i = low;
  while (i < high-1)
    { int t;
      k = minloc (a,i,high);
      t =a[k];
      a[k] = a[i];
      a[i] = t;
      i = i + 1;
    }
}

void main (void)
{ int i;
  i = 0;
  while (i < 10)
    { x[i] = input;
      i = i + 1;
```

```
  sort (x,0,10);
  i = 0;
  while (i < 10)
    { output(x[i]);
      i = i + 1;
}
```

## A.4 C－语言的Tiny Machine运行时环境

下面的描述采用了8.7节给出的Tiny Machine知识和第7章基于栈的运行时环境的知识。因为C－(与TINY不同)有递归过程，运行时环境必须是基于栈的。环境的组成部分有在 dMem顶部的全局区和在它下面的栈，朝下向 0增长。因为C－不包含指针或动态分配，因此就不需要堆(heap)。在C－中每个活动记录(或栈结构)的组成如下




这里，fp是当前结构指针(current frame pointer)，为便于访问保存在一个寄存器中。ofp(旧结构指针)是正文第7章中讨论的控制链(control link)。在FO(结构偏移)右端的常数是每个存储的指示值的偏移量。值 initFO是在一个活动记录中存储区开始的参数和变量的偏移量。因为Tiny Machine不包含栈指针，对活动记录中所有字段的引用都使用带负结构偏移的 fp。

例如，如果有下列C－函数声明：

```
int f(int x, int y)
{ int z;
  ...
}
```

那么x、y和z必须在当前结构中分配，f程序体代码产生的结构起始偏移量是- 5(x、y和z各占一个地址，活动记录的簿记信息占两个地址)。x、y和z的偏移分别是- 2、 -3和-4。

在存储器中全局引用可以用绝对地址找到。然而，像 TINY一样，我们更愿意从一个寄存器的偏移量引用这些变量。通过保存一个固定的寄存器实现这一点，称作 gp，它总是指向最大的地址。因为TM模拟器在执行开始之前把这个地址存储到地址0，启动时gp可以从地址0装入，下面是初始化运行时环境的标准开始序列：

```
0: LD gp,   0(ac)   * load gp with maxaddress
1: LDA fp,  0(gp)   * copy gp to fp
2: ST ac,   0(ac)   * clear location 0
```

函数调用也要求在一个调用序列中使用函数体的开始代码地址。我们也希望使用 pc的当前值执行相对转移来调用函数而不是直接转移 (这将使代码潜在地可重定位 )。程序code.h/

**code.c**中的实用过程**emitRAbs**可以用于这个目的(它接受绝对代码地址，并通过使用当前的代码产生地址使其相对化)。

例如，假设要调用一个函数，其代码起始地址是27，当前的地址是42。那么代替产生绝对转移

```
42: LDC pc, 27(*)
```

我们将产生

```
42: LDA pc, -16(pc)
```

这是因为27 - (42 + 1) = -16。

1) **调用序列** 调用者和被调用者之间的合理划分是：使调用者除了在 retFO地址存储返回指针外，还在新的结构中存储参数的值并创建新的结构。代替存储返回指针本身，调用者把它留在ac寄存器中，被调用者把它存储进新的结构。因此，每个函数体必须从在 (现在当前的)结构中存储值的代码开始：

```
ST ac, retFO(fp)
```

这在每个调用点保存一条指令。在返回时，每个函数通过执行指令

```
LD pc, retFO(fp)
```

用这个返回地址装入pc。相应地，调用者逐个计算参数，在新结构压栈之前把它们压进栈中相应的位置。调用者也必须先把当前的 fp保存进结构的ofpFO处。从被调用者返回后，通过把旧的fp装入fp，调用者丢弃新结构。因此，对有两个参数的函数的调用将产生下列代码：

```
<code to compute first arg>
ST ac, frameoffset+initFO (fp)
<code to compute second arg>
ST ac, frameoffset+initFO-1 (fp)
ST fp, frameoffset+ofpFO (fp) * store current fp
LDA fp, frameoffset(fp) * push new frame
LDA ac,1(pc) * save return in ac
LDA pc, ...(pc) * relative jump to fuction entry
LD fp, ofpFO(fp) * pop current frame
```

2) **地址计算** 因为变量和下标数组都允许出现在赋值表达式的左边，所以在编译期间必须区分地址和值。例如，在语句

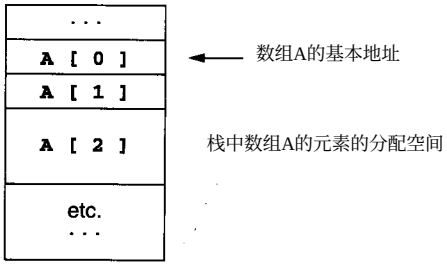
```
a[i] := a[i+1];
```

中，表达式**a[i]**指的是**a[i]**的地址，而表达式**a[i+1]**指的是**a**在地址**i+1**处的值。这个区分可以对**cGen**过程使用一个**isAddress**参数来实现。当这个参数为真时，**cGen**产生的代码计算变量的地址，而不是值。对于简单变量的情况，这意味着加上 gp(全局变量)或fp(局部变量)的偏移量并把结果装入到ac：

```
LDA ac, offset(fp) ** put address of local var in ac
```

对于数组变量的情况，这意味着加上相对于数组基地址的索引值，并把结果装入到 ac，如下所述。

3) **数组** 在栈中数组的分配从当前结构偏移量开始，按下标增长的顺序在存储器中向下延伸，如下所示：




注意，数组的地址通过从基地址中减去索引值计算。

当一个数组传递给函数时，仅传递基地址。基元素区域的分配只进行一次，并在数组生存期间保持固定。函数参数不包括数组的实际元素，仅仅是地址。因此，数组参数是引用参数。当数组参数在函数内部引用时这将引起异常，因为在存储器中保存的必须看成是它们的基地址而不是值。因此，数组参数计算基地址时使用LD操作代替LDA。

## A.5 使用C－和TM的编程设计

基于本书中讨论的TINY编译器(其清单在附录B中)，对于一个学期编译课程来说，要求把一个C－语言的完整的编译器作为设计不是没有道理。这可以进行一些调整，当研究了相关的理论后实现编译器的每个阶段。另一方面， C－编译器的一个或多个部分可以由导师提供，要求学生完成剩余的部分。当时间较短(如1/4学年)或者学生要产生“实际”机器的汇编代码，如Sparc或PC(在代码生成阶段要求更多的细节)，这就特别有用。对于仅实现C－编译器的一部分这就不怎么有用，因为各部分之间的相互作用和代码测试的能力被限制了。下列分列的任务清单提供了一种安排，要注意每个任务与其他任务都不是独立的，最好完成所有的任务以获得完整的编写编译器的经验。

### 设计

1. 实现适合于C－的一个符号表。要求表结构结合作用域信息，用于当各个独立的表链接到一起，或者有一个删除机制，用基于栈的方式操作，如第6章所述。
2. 实现一个C－扫描器，或者像DFA用手工进行，或者使用Lex，如第2章所述。
3. 设计一个C－语法树结构，适合于用分析器产生。
4. 实现一个C－分析器(这要求一个C－扫描器)，或者使用递归下降用手工进行，或者使用Yacc，如第4、5章所述。分析器要产生合适的语法树(见设计3)。
5. 实现C－的语义分析器。分析器的主要要求是，除了在符号表中收集信息外，在使用变量和函数时完成类型检查。因为没有指针或结构，并且仅有的基本类型是整型，类型检查器需要处理的类型是空类型、整型、数组和函数。
6. 实现C－的代码产生器，其根据是前一节描述的运行时环境。