UCC 内部实现

UCC内部实现 1
第一节 体系结构 3
第二节 内存管理 4
2.1 数据结构 4
2.2 接口 5
第三节 类型子系统 5
3.1 数据结构 5
3.2 接口 7
第四节 词法分析 7
4.1 输入处理 8
4.2 接口 8
4.3 标识符和字符串 8
第五节 语法分析 8
5.1 数据结构 9
5.2 C语言语法 10
第六节 语义分析 11
6.1 break, case和continue语句 11
6.2 标号 11
6.3 声明中的类型推导 12
6.4 变量初始化 13
第七节 中间代码生成 13
7.1 数据结构 14
7.2 中间语言语法 15
第八节 中间代码优化 15
8.1 代数简化 15
8.2 值-编号 15
8.3 无用代码消除 16
8.4 窥孔优化 16
8.5 控制流简化 16
第九节 目标代码生成 16
9.1 指令模板 16
9.2 寄存器分配 17
9.3 调用规范 17

ucc 是一个遵从 ANSI C 标准的 C 编译器，在 ucc 的设计和实现过程中，代码结构的清晰性，可扩展性和效率被放在了同等重要的位置。本文档主要讲解 ucc 的实现，帮助开发者更好的掌握源码。

第一节　体系结构

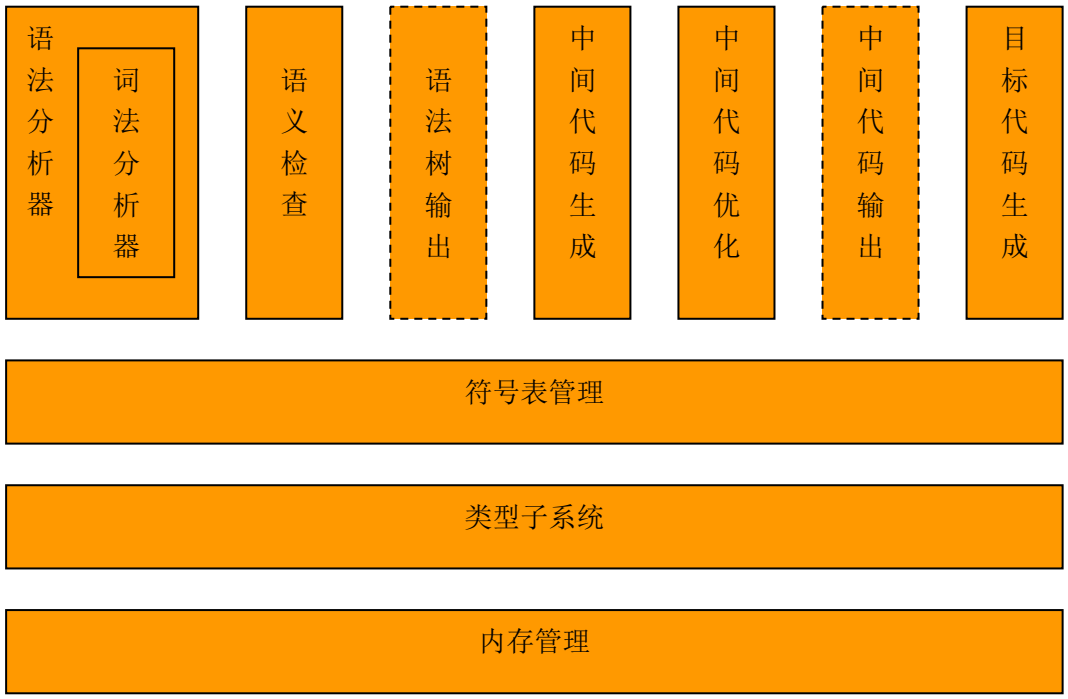


图 1　　ucc 体系结构

对于一个给定的经过预处理的 C 程序，ucc 完成以下几个阶段的处理：

(1) 语法分析：语法分析和词法分析不是两个独立的阶段，在语法分析器的运行过程中，直接调用词法分析器，语法分析的结果是生成语法树。
(2) 语义检查：语义检查直接对语法树进行操作，检查程序语义的正确性。
(3) 语法树输出：图中该阶段使用的是虚线，表明这是一个用户可选的功能。
(4) 中间代码生成：将语法树翻译成中间代码，ucc 使用三地址码。
(5) 中间代码优化：消除中间代码中不必要的部分或者冗余的计算等。
(6) 中间代码输出：与语法树输出一样，这也是一个用户可选的功能。
(7) 目标代码生成：生成目标平台的汇编码。

以上 7 个阶段需要依赖于几个基础模块，其中从下往上依次是内存管理，类型子系统和符号表管理。以上的各个阶段和模块会在以下的章节中详细阐述。

下面按字母顺序列出每个 C 文件的功能：

alloc.c：　　内存管理
ast.c：　　语法分析公用函数
decl.c：　　C 语言声明语法分析

declchk.c： C 语言声明语义检查
dumpast.c： 输出语法树
emit.c： 目标代码生成
error.c： 错误报告
expr.c： C 语言表达式语法分析
flow.c： 中间代码控制流优化
fold.c： 常量折叠
gen.c： 中间代码生成
input.c： 输入处理
lex.c： 词法分析器
output.c： 文件输出
reg.c： 寄存器分配
simp.c： 中间代码简化
stmt.c： C 语言语句语法分析
stmtchk.c： C 语言语句语义检查
str.c： 字符串和标识符处理
symbol.c： 符号表
tranexpr.c： C 语言表达式翻译
transtmt.c： C 语言语句翻译
type.c： 类型子系统
ucl.c： 程序主入口点
uildasm.c： 中间代码输出
vector.c： 动态数组
x86.c： Linux 和 Windows 共有的汇编码生成
x86linux.c： Linux 平台汇编码生成
x86win32.c： Windows 平台汇编码生成

第二节　内存管理

内存管理对于一个编译器的高效运行是非常重要的，ucc 不是简单的使用 malloc 和 free 来完成对内存的分配和释放。由编译器所分配的很多数据结构具有非常重要的一个特性，就是生命期。比如说对于语法树，当释放根节点的时候，相应的子结点也要被释放。因此 ucc 采取了一个基于堆的分配策略。堆是一个可以动态增长的内存块列表。不同于 free 操作，对于堆的释放并不真的释放内存，系统中会维护一个空闲内存块列表，当释放堆时，就将堆中的所有内存块放入空闲内存块列表中。

2.1 数据结构

```
struct mblock
{
    struct mblock *next;
    char *begin;
```

```
    char *avail;
    char *end;
};
```

每一个内存块的头部是一个 mblock 结构，它用以描述一个内存块，ucc 在分配内存时，会预先分配较大的内存块以满足后续的内存分配需求。

- next：　指向堆中下一个内存块的指针
- begin：　内存块的起始
- end：　内存块的结束
- avail：　当前可用的内存

```
typedef struct heap
{
    struct mblock *last;
    struct mblock head;
} *Heap;
```

heap 用以描述一个堆。

- last：　指向堆中最后一个内存块
- head：　内存块列表的头部。

2.2 接口

`void InitHeap(Heap hp)`

初始化堆的内存块列表为空列表。

`void* HeapAllocate(Heap hp, int size)`

从堆中分配 size 大小的内存，如果堆中最后一个内存块不能满足要求，则从空闲内存块列表中寻找一个能满足要求的内存块，如果找不到，则分配一个新的内存块。

`void FreeHeap(Heap hp)`

将堆中所有的内存块回收到空闲内存块列表 FreeBlocks

第三节　类型子系统

C 语言是一个强类型语言，类型子系统实现了 C 语言的所有类型相关操作。

3.1 数据结构

```
#define TYPE_COMMON  \
    int categ : 8;   \
    int qual : 8;    \
    int align: 16;   \
    int size;        \
```

```
    struct type *bty;

typedef struct type
{
    TYPE_COMMON
} *Type;
```

在 ucc 中，对于很多数据结构的定义采取了这样一种类似于对象层次定义的方式。首先定义所有类型的公有字段，可以理解为定义了一个基类型。然后定义派生类型的数据结构。

- categ： 类别，C 语言每个类型都有一个类别。
- qual： 类型修饰符(const 或者 volatile)
- align： 类型按多少字节对齐
- size： 类型的大小
- bty： C 语言的类型可以划分为两个大类：基础类型和导出类型。基础类型比如字符型，整型等。导出类型表示从基础类型的基础上推导而成的类型，比如指针类型。bty 表示从该类型推导而来。对于基础类型，bty 为空。

```
typedef struct field
{
    int offset;
    char *id;
    int bits;
    int pos;
    Type ty;
    struct field *next;
} *Field;
```

field 用来表示一个 struct 或 union 中的字段：

- offset： 字段相对于 struct 或 union 起始处的偏移量。
- id： 字段名。
- bits： 位段的位数，对于常规字段，bits 为 0。
- ty： 字段的类型。
- next： 指向下一个字段。

```
typedef struct recordType
{
    char *id;
    Field flds;
    Field *tail;
    int hasConstFld : 16;
    int hasFlexArray : 16;
} *RecordType;
```

对于 C 语言中的 struct 和 union，使用同一个数据结构 recordType 来描述。

- id： struct 或 union 的名字，对于匿名 struct 或 union，id 为空。
- flds： 所有字段列表。
- tail： 内部字段，用以字段列表的构建。

- hasConstFld：是否包含有常量类型的字段。
- hasFlexArra：是否包含灵活数组（即大小为 0 的数组），灵活数组必须是最后一个字段。

```
typedef struct parameter
{
    char *id;
    Type ty;
    int reg;
} *Parameter;
```

parameter 描述函数声明或函数定义中的参数信息。

- id： 参数名，可以为空。
- ty： 参数类型。
- reg：是否被 register 修饰

```
typedef struct signature
{
    int hasProto : 16;
    int hasEllipse : 16;
    Vector params;
} *Signature;
```

signature 描述函数签名。

- hasProto： 是否函数原型。
- hasEllipse： 是否拥有可变参数。
- params： 参数集合。

C 标准允许旧式风格的函数声明和定义，新式风格要求函数有原型。对于旧式风格的函数定义，虽然没有函数原型，但是仍然具备参数声明。因此在 signature 中 hasProto 为 0 的情况下，params 可能不为空。

3.2 接口

类型子系统提供了一些断言宏，比如 IsIntegType, IsPtrType 等；还有一些类型构造函数，比如 PointerTo 构造指针类型等；还有其它一些函数。

`void SetupTypeSytem(void)`

设置类型系统，主要是设置基础类型的大小，对齐等。该函数依赖于 config.h 中的定义，对于不同的平台，类型的表示是不一样的。

第四节 词法分析

经过预处理后的 C 文件可以看作是一个字符流，词法分析就是把字符流变成记号流。记号是对一个或多个字符做的一个基本归类，比如说标识符。C 语言中的记号有字符串，常量，关键字，标识符和标点符号。词法分析有一个基本的原则：最长匹配原则。比如说碰到++，词法分析器会识别为自增操作符而不是加号。

4.1 输入处理

ucc 直接利用操作系统提供的内存映射文件机制来将整个文件映射到内存中，文件被看作是一个无符号字符数组。并在数组的最后追加一个特殊字符 END_OF_FILE（值为 255）。这样当词法分析器遇到这个字符时，就认为文件已经结束。

4.2 接口

记号用整型来表示，记号的定义在文件 token.h 中。记号可能会有相应的值，比如说每个标识符对应了一个字符串，记号的值被记录在全局变量 TokenValue 中。

`void SetupLexer(void)`
词法分析器是基于一个函数指针数组 Scanners，以无符号字符的值作为索引，每一个无符号字符对应了一个函数。SetupLexer 就是设置该指针数组的每一项为相应的函数指针。

`int GetNextToken(void)`
从当前字符流中返回相应的记号。

`void BeginPeekToken(void)`
`void EndPeekToken(void)`
在做语法分析时，有时需要根据下一个记号来决定相应的操作。提供这一组调用使得语法分析器能够在记号流中做出一个标记，而且能重新回到这个标记。

4.3 标识符和字符串

ucc 对于组成标识符的字符串维护一个字符串池，对于相同的标识符，字符串池保持唯一的一份拷贝。这样使得在比较标识符是否相等时，只需进行简单的指针比较就可以了。对于字符串，并没有保持唯一的拷贝。

第五节　语法分析

语法分析分为三个部分：声明，表达式和语句。ucc 的语法分析有几个很重要的规则。

(1) 对于每一个非终结符，有一个相应的解析函数。这些函数遵循同样的命名规则，Parse + 非终结符。比如 ParseIfStatement, ParseTranslationUnit 等。
(2) 在源码中，对于每一个非终结符解析函数在注释中给出了相应的语法产生式，这些语法产生式与 ANSI C 给出的语法产生式几乎是一致的。
(3) 语法分析的结果是构造一个语法树(AST)，语法树的根对应了 C 语言的起始节点 translation-unit。每一个内部节点对应了一个非终结符，叶子节点对应了记号。对于声明和语句中的每一个非终结符定义了相应的数据结构。但表达式例外，所有的表达式用同一个数据结构来表示。

5.1 数据结构

```
#define AST_NODE_COMMON   \
    int kind;             \
    struct astNode *next;  \
    struct coord coord;

typedef struct astNode
{
    AST_NODE_COMMON
} *AstNode;
```

AstNode 是语法树中所有节点的基节点。

- kind： 节点的类别，ast.h 中定义了枚举类型 nodeKind 列出了所有的节点类别。
- next:： 指向下一个节点，用于构造节点列表。
- coord： 节点所在文件中的行和列信息。

对于声明和语句的每一个非终结符，定义一个相应的数据结构。结构的字段对应了非终结符产生式的右端的每一项。比如：

if-statement:

if (expression) statement

if (expression) statement else statement

相应的数据结构为

```
AstIfStatement
{
    AST_NODE_COMMON
    AstExpression expr;
    AstStatement thenStmt;
    AstStatement elseStmt;
};

struct astExpression
{
    AST_NODE_COMMON
    Type ty;
    int op : 16;
    int isarray : 1;
    int isfunc  : 1;
    int lvalue  : 1;
    int bitfld  : 1;
    int inreg   : 1;
    int unused  : 11;
    struct astExpression *kids[2];
    union value val;
};
```

astExpression 定义了表达式节点。

- ty：　表达式的类型
- op：　表达式的操作符
- isarray：是否是数组
- isfunc：是否是函数
- lvalue：是否是左值
- bitfld：是否是位段
- inreg：是否被声明为寄存器变量
- kids：　表达式的操作数。
- val：　表达式的值

5.2 C 语言语法

在大多数情况下，C 语言是一个 LL(0)文法，解析器可以根据当前的记号而决定与什么产生式和非终结符匹配。但是由于 C 语言引入了类型定义而破坏了这个特性。比如说在复合语句中遇到一个标识符，如果是类型定义名，则应该为声明，否则为语句。为了简洁性，ucc 将语法分析和语义检查分成两个独立的阶段，但是类型定义又要求在语法分析中做一定的语义检查，为此，ucc 在语法分析中对于类型定义做最小的检查。对于语法分析器来说，只要知道一个标识符是类型定义名就足够了，而不需要知道具体的类型。

语法分析中用以进行语义检查的数据结构和算法如下：

```
typedef struct tdname
{
    char *id;
    int level;
    int overload;
} *TDName;
```

tdname 表示一个类型定义名。

- id：　名字
- level：　类型定义名被定义点的嵌套层次，文件作用域的嵌套层次为 0，复合语句的起始处嵌套层次加 1，复合语句的结束处嵌套层次减 1。有可能在多个作用域中定义同一个类型定义名，level 表示这些作用域中嵌套层次最小的。比如说对于如下代码片段：

  ```
  typedef int a;
  int f(void)
  {
      typedef int a;
  }
  ```

  其中 level 值应为 0。这样在其它函数中 a 的定义也是可见的。
- overload：表示在嵌套作用域中，类型定义名是否作为变量，而不是类型名。比如如下代码片段：

  ```
  typedef int a;
  int f(int a)
  {
  }
  ```

  在 f 的函数定义中，a 是参数，而不是类型定义名。

ucc 使用两个向量：TypedefNames 记录了所有定义的类型名。OverloadNames 记录当前作用域中被重载了的类型名。

对于每一个声明，调用 CheckTypedefName 函数，如果该声明属于类型定义，则查看相应的类型名是否存在，如果不存在，则加入新的类型定义，如果存在，修改 level 为最小的嵌套层次。如果该声明不属于类型定义并且该声明所定义的变量已定义为外部作用域的类型名，则标记该类型名被重载，并且将其加入 OverloadNames 中。

每当一个复合语句结束时，重置 OverloadNames 中的所有类型名的重载状态。

第六节　语义分析

在语法分析结束后，要分析所给定的程序是否满足 C 语言规定的语义。在 ANSI C 标准中，对于每一个语法结构都有一个相应的 Constraints 节，语义分析几乎是对该节文字描述的代码翻译，因此建议结合 ANSI C 标准文档能够很快的理解这块代码。接下来重点介绍一下语义分析中的几个关键部分。

6.1 break, case 和 continue 语句

对于 break，必须有对应的最内层的循环或 switch 语句。对于 case，必须有对应的最内层的 switch 语句。对于 continue，必须有对应的最内层的循环语句。在语法树的函数节点 astFunction 中维护三个堆栈: loops（所有循环），swtchs（所有 switch 语句）和 breakable（所有循环和 switch 语句）。栈顶是最内层的循环或 switch 语句。

6.2 标号

C 语言中的标号是作用域是整个函数，标号的引用可以在标号的定义之前。

```
typedef struct label
{
    struct coord coord;
    char *id;
    int ref;
    int defined;
    BBlock respBB;
    struct label *next;
} *Label;
```

label 用以表示函数中的标号。

- coord：　标号的坐标，这样当函数结束时，可以检查未定义的标号，报告错误。
- id：　　标号名
- ref：　　标号的引用次数。
- defined：标号是否被定义。

- respBB：标号对应的基本块，中间代码生成使用该字段。
- next：　下一个标号。

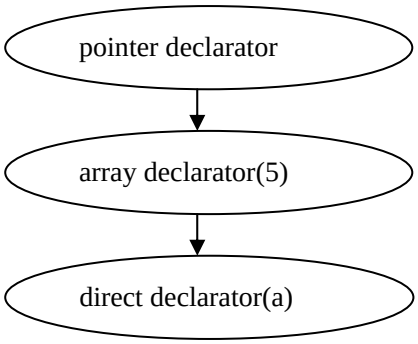
6.3 声明中的类型推导

C 语言的设计希望对于变量的类型声明和使用时保持一致。比如

```
int a[5]；
a[1] = 3;
```

而不是像其他语言那样：int[5] a;这种设计在一定程度上带来了 C 语言类型声明的复杂性。
ucc 中对于类型推导使用了非常方便的处理方式：

比如说 int *a[5];



在语法分析时生成如上图所示的语法树，声明的结合顺序与操作符的优先级有类似之处。
标识符自身的优先级最高，属于 direct declarator。
数组和函数的优先级相同，分别属于 array declarator 和 function declarator。
指针的优先级最低，属于 pointer declarator。
每一个 declarator 都具有一个属性，定义如下：

```
typedef struct typeDerivList
{
    int ctor;
    union
    {
        int len;
        int qual;
        Signature sig;
    };
    struct typeDerivList *next;
} *TypeDerivList;
```

typeDerivList 描述类型推导列表。

- ctor：类型构造符，可以为数组，函数，指针。
- len：　表示数组长度。
- sig：　表示函数签名。
- qual：表示指针限定符。
- next：指向下一个类型推导列表。

在语义检查过程中，ucc 自底往上构造这个类型推导列表。对于上面的语法树，该类型推导

列表为(pointer to) → (array of 5) → NIL

`Type DeriveType(TypeDerivList tyDrvList, Type ty)`
该函数对基类型 ty 从左至右依次应用类型推导列表中的类型推导，满足了类型构造符的优先级顺序。对于 int *a[5]，类型推导的结果为 array of 5 pointer to int。

6.4 变量初始化

ucc 对于变量的初始化采取一致的处理，任何变量看作是一个字节数组。

```
struct initData
{
    int offset;
    AstExpression expr;
    InitData next;
};
```

initData 描述了一块内存的初始化。

- offset：相对于变量内存起始处的偏移
- expr： 用以初始化内存的表达式，表达式的类型指定了内存大小
- next： 下一个初始化数据

比如说对于如下结构的初始化：

```
struct st
{
    struct
    {
        int a, b;
    };
    int c;
} st1 = {{2}, 3};
```

分析后的结果为(offset:0, expr:2) → (offset:8, expr:3)。b 会被初始化为 0。

第七节　中间代码生成

ucc 采用三地址码作为中间代码，同时构建由基本块组成的控制流图。
一个基本块是一个线性代码序列，在执行时只有一个入口点和一个出口点。
比如对于如下代码段：

```
a = 3;
b = 4;
if a > b goto L1;
b = 5;
L1:
b = 3;
```

上面的代码由三个基本块组成：

```
a = 3; b = 4; if a > b goto L1
```

```
b = 5;
L1: b = 3;
```

7.1 数据结构

```
typedef struct irinst
{
    struct irinst *prev;
    struct irinst *next;
    Type ty;
    int opcode;
    Symbol opds[3];
} *IRInst;
```

irinst 给出了中间语言的指令定义。

- prev：　指向上一条指令
- next：　指向下一条指令
- ty：　指令操作类型
- opcode：操作码
- opds：　指令操作数，最多三个操作数。

```
struct bblock
{
    struct bblock *prev;
    struct bblock *next;
    Symbol sym;
    CFGEdge succs;
    CFGEdge preds;
    struct irinst insth;
    int ninst;
    int nsucc;
    int npred;
};
```

bblock 给出了基本块的定义。

- prev：　指向上一个基本块
- next：　指向下一个基本块
- sym：　代表基本块的符号
- succs：　该基本块所有的后继
- preds：　该基本块所有的前驱
- insth：　该基本块的指令列表
- ninst：基本块中的指令数目
- nsucc：后继个数
- npred：前驱个数

7.2 中间语言语法

Program	→	IRInst*
IRInst	→	AssignInst | BranchInst | JumpInst | IJumpInst |
		ReturnInst | CallInst | ClearInst
AssignInst	→	VarName = Expression |
		VarName = Operand |
		*VarName = Operand
BranchInst	→	If Operand RelOper Operand goto Label |
		If Operand goto Label |
		If ! Operand goto Label
JumpInst	→	goto Label
IJumpInst	→	goto (label1, label2, …)[VarName]
ReturnInst	→	return VarName
CallInst	→	[VarName =] call VarName(Operand, Operand, …)
ClearInst	→	Clear VarName, constant
BinOper	→	| ^ & + - * / %
UnaryOper	→	* & - ~
Label	→	identifier
VarName	→	identifier
Operand	→	VarName | constant

第八节　中间代码优化

ucc 对生成的中间代码做了一些最基本的优化。

8.1 代数简化

代数简化是应用一些基本的代数公式，比如 a + 0 = a, a * 1 = a 等；还有将一些复杂的操作转换成简单的操作：比如说 a * 8 转换成 a >> 3 等。

8.2 值-编号

ucc 进行局部的值编号，即只在基本块内部进行。值编号的目的是为了避免重复计算：
比如以下代码:

```
t1 = a + b;
t2 = a + b;
```

可以变为:

```
t1 = a + b;
t2 = t1;
```

其中 a + b 的值被编号为 t1。

8.3 无用代码消除

在中间代码的生成过程中，会产生一些从未被使用的临时变量定义，无用代码消除就是找到这些未被使用的临时变量，删除对应的指令。

8.4 窥孔优化

这部分优化包括识别 a = a + 1 这样的指令将其变成对 a 的自增操作。
还有比如对于 a = f(1)产生的中间代码为：
t = f(1); a = t;
将其合并为 a = f(1)。

8.5 控制流简化

在中间代码生成过程中，会产生一些空基本块或者一些直接到下一个基本块的跳转。
控制流简化主要就是为了合并这些基本块。

第九节　目标代码生成

ucc 实现了一个非常简单的代码生成器，将每一个中间代码指令翻译成一个或多个目标代码指令。ucc 目前生成 Linux 和 Windows 平台下的 x86 汇编码。其中公用的代码实现在 x86.c 中，这主要是对中间代码的翻译。由于 Linux 和 Windows 平台下的汇编器要求不同的汇编格式，因此定义一组接口，x86linux.c 和 x86win32.c 提供了对这一组接口的不同实现。

9.1 指令模板

对于每一个中间代码操作码和相应的操作类型定义了一个对应的目标代码操作码，每个目标代码操作码有一个相应的指令模板。x86linux.tpl 和 x86win32.tpl 提供了相应的指令模板。所有的指令模板构成了一个指令表。

`void PutASMCode(int code, Symbol opds[])`

该函数将 code 对应的指令模板用相应的操作数代替，输出汇编码。
比如下面的指令模板：
or %0, %d1;
其中%表示转义符，其后的字符需要进行解释，其他字符原样输出。
可能出现在转义符后的字符有：

(1) 数字：　只能是 0，1 和 2。分别表示第 1，第 2 和第 3 个操作数。
(2) %：　输出%。
(3) b, w, d：　很多汇编语言在指令中编码操作数类型。但 MASM 使用关键字来指定操作数类型。b, w, d 分别对应了 BYTE PTR, WORD PTR 和 DWORD PTR。

9.2 寄存器分配

x86 对于整型数据和浮点型数据采取不一样的寄存器结构。
ucc 只对临时变量分配寄存器。整型数据的寄存器分配使用龙书中介绍的最简单的寄存器分配算法。浮点型数据的寄存器分配只使用 ST0 和 ST1 两个寄存器，并保证进入每一个基本块时，浮点寄存器堆栈处于初始状态。

9.3 调用规范

为了保持与 gcc 和 vc 编译器的二进制兼容性，ucc 遵从这些编译器的调用规范。

(1) 函数调用时，实参从右往左进栈。调用者负责实参出栈。

(2) 保值寄存器：EBX, ESI 和 EDI。这些寄存器由被调用者保存。
易失寄存器：EAX, ECX 和 EDX。这些寄存器由调用者保存。

(3) 函数返回值的处理：
若返回值为整型，返回值位于 EAX 寄存器。
若返回值为浮点型，返回值位于 ST(0)寄存器。
若返回值为大小为 1，2 或 4 的 struct/union 类型，返回值位于 EAX 寄存器。
若返回值为大小为 8 的 struct/union 类型，返回值位于 EAX,EDX 寄存器。
若返回值为其它大小的 struct/union 类型，函数的第一个参数是隐式参数，该参数为函数返回值接收变量的地址，函数的返回类型成为 void。