UCC Internal

Wenjun Wang(wenjunw@yahoo.cn)

UCC Internal ... 1
- Section1 Architecture ... 3
- Section2 Memory Management ... 4
  - 2.1 Data Structures ... 5
  - 2.2 Interfaces ... 5
- Section3 Type subsystem ... 5
  - 3.1 Data Structures ... 6
  - 3.2 Interfaces ... 7
- Section4 Lexical Analyzer ... 8
  - 4.1 Input processing ... 8
  - 4.2 Interface ... 8
  - 4.3 Identifier and String ... 8
- Section5 Syntax Parser ... 9
  - 5.1 Data Structures ... 9
  - 5.2 C language grammar ... 10
- Section6 Semantic Check ... 11
  - 6.1 break, case and continue ... 11
  - 6.2 Label ... 12
  - 6.3 Type derivation ... 12
  - 6.4 Initialization ... 13
- Section7 Intermediate code generation ... 14
  - 7.1 Data Structures ... 14
  - 7.2 Intermediate language Grammar ... 15
- Section8 Intermediate code optimization ... 16
  - 8.1 Algebraic Simplification ... 16
  - 8.2 Value-numbering ... 16
  - 8.3 Unused code elimination ... 16
  - 8.4 Peephole ... 16
  - 8.5 Control flow simplification ... 16
- Section9 Target Code generation ... 17
  - 9.1 Instruction Template ... 17
  - 9.2 Register Allocation ... 17
  - 9.3 Calling Convention ... 17

UCC is a ANSI C compiler. The clarity and extensibility was as important as the efficiency during the design and implementation of ucc. This document mainly illustrates the design and implementation of ucc to help developers master the source code.

Section1 Architecture

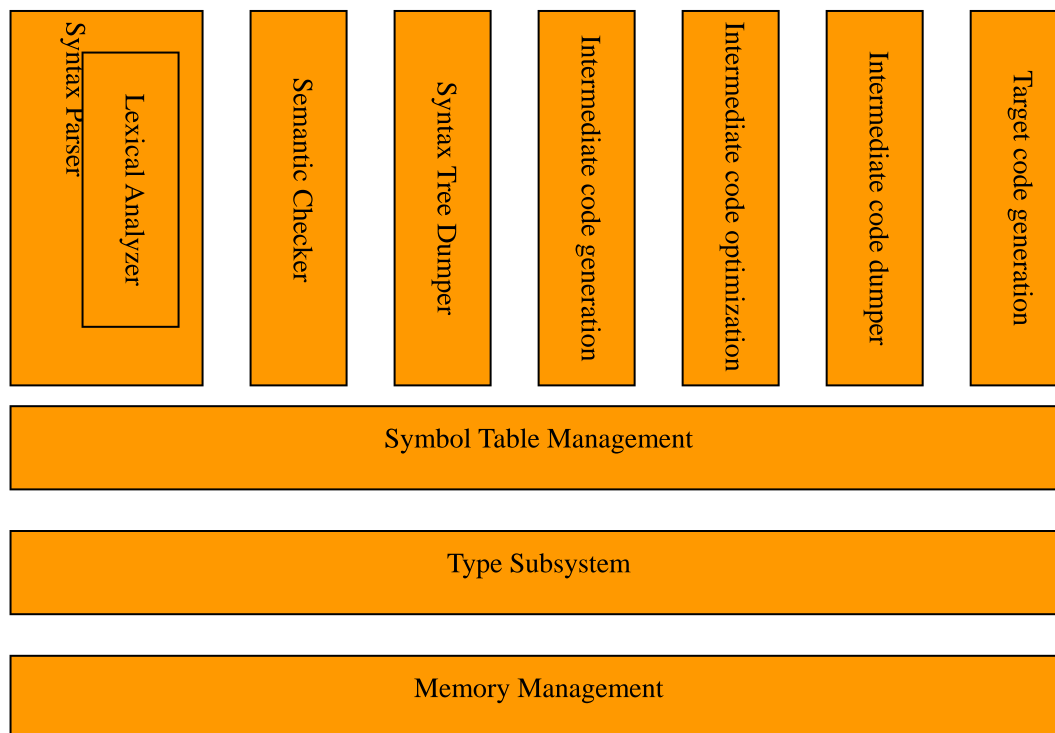


Figure 1 UCC Architecture

For a given preprocessed C source file, ucc will process it through the following stages:

(1) Syntax Parser: The syntax parser and the lexical analyzer is not separate stage, the syntax parser will directly use the lexical analyzer. The result of syntax parsing is to produce a syntax tree.

(2) Semantic Checker：The semantic checker operates on the syntax tree, check the correctness of the program's semantic.

(3) Syntax tree Dumper: Dump the syntax tree. In above figure using dash line for this stage to indicate it is an optional functionality.

(4) Intermediate code generation：Translate the syntax tree into the intermediate code, ucc uses three address code as intermediate code.

(5) Intermediate code optimization：Eliminate unused code or redundant computation etc.

(6) Intermediate code dumper：Same as syntax tree dumper, this is also an optional functionality.

(7) Target code generation: Generate the assembly code for target platform.

The above seven stages depend on several foundation modules which are memory management, type subsystem and symbol table management from the bottom up. We will elaborate on these

stages and modules in the following sections.

Here is the list of each C file's functionality in alphabetical order:

alloc.c： memory management
ast.c： general routines used by syntax parser
decl.c： C declaration syntax parsing
declchk.c： C declaration semantic check
dumpast.c： syntax tree dumper
emit.c： target code generation
error.c： error reporting
expr.c： C expression syntax parsing
flow.c： intermediate code control flow optimization
fold.c： constant folding
gen.c： intermediate code generation
input.c： input processing
lex.c： lexical analyzer
output.c： output
reg.c： register allocation
simp.c： intermediate code simplification
stmt.c： C statement syntax parsing
stmtchk.c： C statement semantic check
str.c： string and identifier processing
symbol.c： symbol table
tranexpr.c： C expression translation
transtmt.c： C statement translation
type.c： type subsystem
ucl.c： program's main entry point
uildasm.c： intermediate code dumper
vector.c： dynamic array
x86.c： assembly code generation shared by Linux and Windows
x86linux.c： Linux specific assembly code generation
x86win32.c： Windows specific assembly code generation

Section2 Memory Management

Memory management is vital for a compiler's efficiency. UCC doesn't simply use malloc and free to allocate and free memory. The data structures used by the compiler have a very important property: lifetime. For example, when freeing the root node of a syntax tree, all the children nods will be freed. So ucc adopts a memory management strategy based on the heap. The heap is a dynamically expanded list of memory block. Different with free, when freeing the heap, the memory will not actually be free. The system will maintain a free memory block list, all the memory blocks will put into this list when freeing the heap.

2.1 Data Structures

```
struct mblock
{
    struct mblock *next;
    char *begin;
    char *avail;
    char *end;
};
```

The header of each memory block is the struct mblock which is used to describe a memory block. Per allocation, ucc will allocate a much bigger memory block in advance to satisfy the successive requirement.

- next: pointer to next memory block in the heap
- begin: beginning of memory block
- end: end of memory block
- avail: currently available memory

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

struct heap is used to describe a memory heap.

- last: pointer to last memory block in the heap
- head: memory block list head

2.2 Interfaces

`void InitHeap(Heap hp)`

Initializes the heap's memory block list as an empty list.

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

Allocate size bytes and returns a pointer to the allocated memory. If the last memory block in the heap can't satisfy the request, find a big enough memory block from the free memory block list. If not found, create a new memory block.

`void FreeHeap(Heap hp)`

Recycle all the memory blocks of the heap into free memory block list.

Section3 Type subsystem

C is a strong type language, the type subsystem implements all the type relevant operations.

3.1 Data Structures

```
#define TYPE_COMMON  \
    int categ : 8;   \
    int qual : 8;    \
    int align: 16;   \
    int size;        \
    struct type *bty;

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

In ucc, many data structures are defined in a manner similar as object hierarchy. At first define all the common field, you can understand it as defining a base type. Then define all the derived type.

- categ: category, each type will have a category.
- qual: type qualifier(const or volatile)
- align: type alignment
- size: type size
- bty: The C language's type can be divided into two kinds: primary type and derived type. Primary type is char, int etc. Derived type is derived from another type, e.g. pointer type. bty is the deriving type. For primary type, bty is NULL.

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

Field represents a field in struct/union type.

- offset: offset to the beginning of struct/union
- id: field name
- bits: bits of a bit field. For non-bit field, bits is 0
- ty: field type
- next: pointer to next field

```
typedef struct recordType
{
    char *id;
    Field flds;
    Field *tail;
```

```
    int hasConstFld : 16;
    int hasFlexArray : 16;
} *RecordType;
```

We use same data structure RecordType for both struct and union type.

- id: struct/union name, for anonymous struct/union, id is NULL.
- flds: all the fields of struct/union
- tail: used for construction of field list
- hasConstFld: contains constant field or not
- hasFlexArray: contains flexible array(array with size 0) or not, the flexible array must be last field

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

Parameter describes the parameter information in function declaration or definition.

- id: parameter name, can be NULL
- ty: parameter type
- reg: qualified by register or not

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

Signature describes the function signature.

- hasProto: has function prototype or not
- hasEllipse: has variable argument or not
- params: parameter set

The C standard permits old-style function declaration and definition. New-style required that each function has prototype. For some old-style functions, although they don't have prototype, they have parameter declaration. So when hasProto is 0 in signature, params maybe not NULL.

3.2 Interfaces

Type subsystem provides some predicate marocs, e.g. IsIntegType, IsPtrType etc; and some type construction functions, e.g. PointerTo to construct pointer type etc; and some other functions.

`void SetupTypeSytem(void)`

Setup the type system, mainly set the size, alignment of primary type. This function depends on the definition at config.h. For different target platform, the type's representation is different.

Section4 Lexical Analyzer

The preprocessed C source file can be viewed as a byte stream. Lexical analysis is to turn the byte stream into token stream. Token is a basic classification for one or multiple characters, e.g. identifier. The token in the C language is string, constant, keyword, identifier and punctuator. There is a basic principle in lexical analyzer: longest match principle. For example if encountering ++, it will be recognized as an increment operator instead of a plus.

4.1 Input processing

ucc directly utilize the memory mapped file mechanism provided by underlying OS to map the file into memory. A file is viewed as an unsigned char array, ucc will append a special character END_OF_FILE (value is 255) to the array. In this way, when lexical analyzer sees this character, the file is ended.

4.2 Interface

The token is represented by int. All the token definitions locate at token.h. A token may have value, e.g. each identifier is accompanied with a string. TokenValue records the token's value.

`void SetupLexer(void)`

Lexical analyzer is based on a function pointer array Scanners. Using the value of unsigned character as index, each unsigned character corresponds to a function. This function will set each element of this array as the corresponding function pointer.

`int GetNextToken(void)`

Returns token from current character stream.

`void BeginPeekToken(void)`
`void EndPeekToken(void)`

When doing syntax parsing, sometime next token is needed to decide how to do. These functions make syntax parser can mark the token stream and go back to the mark.

4.3 Identifier and String

ucc maintains a string pool for the identifiers. For same identifier, there is a unique copy in this pool which causes identifier comparison is very simple, just pointer comparison. There is not a unique copy for string.

Section5 Syntax Parser

Syntax parsing is divided into three parts: declaration, expression and statement. There is several important rules in ucc's syntax paring:

(1) There is a function for each nonterminal. These functions obey same naming rules, Parse + nonterminal, e.g. ParseIfStatement, ParseTranslationUnit.

(2) In the source code, the comments for each nonterminal parsing function give the production rule. These grammar production rules are almost identical as those given by ANSI C.

(3) The result of syntax parsing is an abstract syntax tree (AST). The root of this tree is the start node: translation-unit. Each internal node represents a nonterminal, leaf node represents a token. There is a data structure definition for each nonterminal in declaration and statement. But all the expressions are described by same data structure.

5.1 Data Structures

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

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

AstNode is the base node of all the nodes in abstract syntax tree.

- kind: the kind of node
- next:: pointer to next node
- coord: the coordinate of node

For each nonterminal data structure definition of declaration and statement, the fields correspond to each item in the right hand side of the production rule for this nonterminal. For example,
if-statement:

    if (expression) statement
    if (expression) statement else statement

the corresponding data structure is

```
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 defines expression node.

- ty: expression type
- op: expression operator
- isarray: array or not
- isfunc: function or not
- lvalue: lvalue or not
- bitfld: bit-field or not
- inreg: declared as register variable or not
- kids: expression operands
- val: expression value

5.2 C language grammar

In most cases, C language is a LL(0) grammar, the parser can decide which production or nonterminal will be matched according to current token. But since C introduces the typedef, this feature is inexistent. For example if encounter an identifier in the compound statement, if it is a typedef name, a declaration is expected, otherwise a statement is expected. For clarity, ucc separate the syntax paring and semantic check in two stages. But typedef requires necessary semantic check during syntax paring. In order to handle this, ucc will perform minimum check for typedef. Knowing that if an identifier is a typedef name is enough for the syntax parser, no need to know the concrete type.

The data structures and algorithm to perform such check are:

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

tdname represent a typedef name.

- id：	name
- level：	the nesting level of the definition point. The nesting level of file scope is 0. The beginning of a compound statement will increment nesting level; the end of a compound statement will decrement nesting level. It is possible that there are definitions for same identifier at multiple scope。For example the following code snippet：

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

  The level will be 0, then a is visible in other scope.
- overload：Indicate if the typedef name is used as variable instead of a typedef name in nesting scope. For example the following code snippet:

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

  a is a parameter instead of typedef name in f()'s definition.

ucc uses two vectors: TypedefNames records all the typedef names. OverloadNames records those overloaded typedef names in current scope.

For each declaration, call CheckTypedef(). If the declaration is a type definition, look up if the typedef name exists, if inexistent new typedef name will be added, otherwise modify level to be the minimum nesting level. If the variable defined by this declaration was already defined as a typedef name in outer scope, mark the typedef name as overloaded and add it to OverloadNames.

When a compound statement ends, reset all the typedef name's overload status in OverloadNames.

Section6 Semantic Check

After syntax parsing, semantic check needs to be performed to check if the given program satisfies C language semantic. In ANSI C standard, each grammar construct has a Constraints section, ucc's semantic check is almost the code translation of this section. Accompany with the standard, these code will be easily understood. This section will introduce some critical part.

6.1 break, case and continue

For break, there must be an innermost loop or switch statement. For case, there must be an innermost switch statement. For continue, there must be an innermost loop statement. ucc uses three stacks: loops, swtchs and breakable in function node astFunction. Top of the stack will be the innermost loop or switch statement.

6.2 Label

The scope of label is whole function。The label’s reference can appear before the label’s definition.

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

label describes the function label.

- coord: label’s coordinate.
- id: label’s name
- ref: label’s reference count
- defined: if label is defined
- respBB: corresponding basic block, used by intermediate code generation
- next: pointer to next label

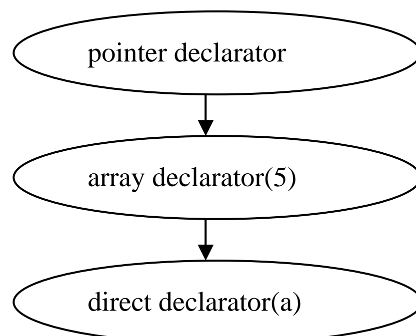
6.3 Type derivation

There is the C philosophy that the declaration of an object should look like its use. For example:

```
int a[5];
a[1] = 3;
```

unlike other languages: int[5] a. The design brings in certain complexity for C declaration.

ucc’s declaration processing is very convenient:
For example: int *a[5];



The above syntax tree will be produced during syntax parsing. The declaration’s binding order is similar as operator’s priority.
Identifier’s priority is highest, belongs to direct declarator。
Array and function’s priority is same， belongs to array declarator and function declarator respectively.

Pointer's priority is lowest, belongs to pointer declartator.
Each declarator owns an attribute defined below:

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

typeDerivList describes a type derivation list.

- ctor：type constructor，can be array, function, pointer.
- len： array length
- sig： function signature
- qual：pointer qualifier
- next：pointer to next type derivation list

During semantic check, ucc will construct the type derivation list from bottom up. For above syntax tree, the type derivation list will be (pointer to) → (array of 5) → NIL

`Type DeriveType(TypeDerivList tyDrvList, Type ty)`

This function will apply the type derivation in the type derivation list from left to right for the base type ty, which in essence is in the order of type constructor's priority. For int *a[5]， the result of type derivation is "array of 5 pointer to int"。

6.4 Initialization

ucc uses a consistent way to handle variable initialization, any variable is viewed as a byte array.

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

initData describes the initialization for a memory block.

- offset：offset to the beginning of the variable memory
- expr： expression，expression's type decides the memory size
- next： pointer to next item

Given the following structure:

```
struct st
{
    struct
```

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

The result will be (offset:0, expr:2) → (offset:8, expr:3)。b will be initialized to 0.

Section7 Intermediate code generation

ucc uses three-address code as intermediate code, at the same time it will construct the control flow graph constituted by basic blocks.

A basic block is a linear code sequence, when executing, it will have only one entry point and one exit point.

Given the following code snippet:

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

There are three basic blocks:

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

7.1 Data Structures

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

irinst gives the intermediate language instruction definition.

- prev： pointer to previous instruction
- next： pointer to next instruction
- ty： instruction operating type
- opcode：operation code
- opds： operands, at most three

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

bblock gives the basic block definition.

- prev: pointer to previous basic block
- next: pointer to next basic block
- sym: symbol to represent basic block
- succs: all the successors of this basic block
- preds: all the predecessors of this basic block
- insth: instruction list
- ninst: number of instructions
- nsucc: number of successors
- npred: number of predecessors

7.2 Intermediate language Grammar

```
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
```

Section8 Intermediate code optimization

ucc performs some basic optimization on the intermediate code.

8.1 Algebraic Simplification

Algebraic simplification will apply some algebraic formulas: such as a + 0 = a, a * 1 =a; and transform some complex operation into simple operation, e.g. a * 8 is turned into a >> 3.

8.2 Value-numbering

ucc performs local value-numbering optimization, i.e. in the basic block. The intention of value-numbering is to avoid repetitive computation.

For example, giving the following code snippet:

t1 = a + b;

t2 = a + b;

The code snippet can be turned into:

t1 = a + b;

t2 = t1;

a + b ‘s value is numbered as t1.

8.3 Unused code elimination

During intermediate code generation, there will be some temporaries never used. Unused code elimination will find these temporaries and remove relevant instructions.

8.4 Peephole

Peephole optimization will recognize some instruction combination and replace those instructions using more convenient instruction. For example,

a = a + 1 will be turned into increment a.

a = f(1)’s intermediate code is t = f(1); a = t; which will be turned into a = f(1).

8.5 Control flow simplification

Intermediate code generation will generate some empty basic blocks or jump to next basic block. Control flow simplification will merge these basic blocks.

Section9 Target Code generation

ucc implements a too simple code generator, translate each intermediate instruction into one or multiple target instructions. Now ucc supports the assembly code on Linux and Windows. The code shared by Linux and Windows is put into x86.c, which is translation to intermediate code. Since the assembler on Linux and Windows requires different format, so a group of interface is defined. x86linux.c and x86win32.c provides specific implementation of this group of interface.

9.1 Instruction Template

For each intermediate instruction's op code and op type there is corresponding target opcode.
Each target op code corresponds to a instruction template. x86linux.tpl and x86win32.tpl provide these instruction templates. All the templates constitute an instruction table.

`void PutASMCode(int code, Symbol opds[])`
This function will parse the instruction template indexed by code, output the assembly.
Given the following instruction template:
or %0, %d1;
Thereof % is used for escape， the character after it will be parsed. Other characters will be copied unchanged to the output.
The characters which can follow % are:
(1) digit： must be 0，1 or 2。Indicate the first, second or third operand.
(2) %： output %
(3) b, w, d： Many assembler encodes the operand type in instruction, but MASM uses keyword to indicate operand type. b, w, d represents BYTE PTR, WORD PTR and DWORD PTR。

9.2 Register Allocation

The organization of integer and float register is different on Intel X86 platform.
ucc only allocates register for temporary. The register allocation for temporary with interger type use the simple register allocator in Dragon book. The register allocation for temporary with float type will only use ST0 and ST1. Per the entry to each basic block, the float register stack will be in the initial status.

9.3 Calling Convention

In order to keep the binary compatibility with gcc and vc, ucc conforms to the calling convention of these compilers.
(1) When calling a function, the actual parameters will be pushed onto stack from right to left. The caller will pop the parameters.
(2) preserve registers：EBX, ESI and EDI which will be saved by the callee.

scratch registers：EAX, ECX and EDX which will be saved by the caller.

(3) function return value：

if the return value is integer type，EAX holds the return value.

if the return value is floating point type，ST(0) holds the return value.

if the return value is struct/union type in size 1, 2 or 4, EAX holds the return value.

if the return value is struct/union type in size 8, (EAX, EDX) holds the return value.

if the return value is struct/union type in other size, the function’s first parameter is implicit which is the address of the function return value’s receiver, the function’s return type will be void.