CLike Compiler

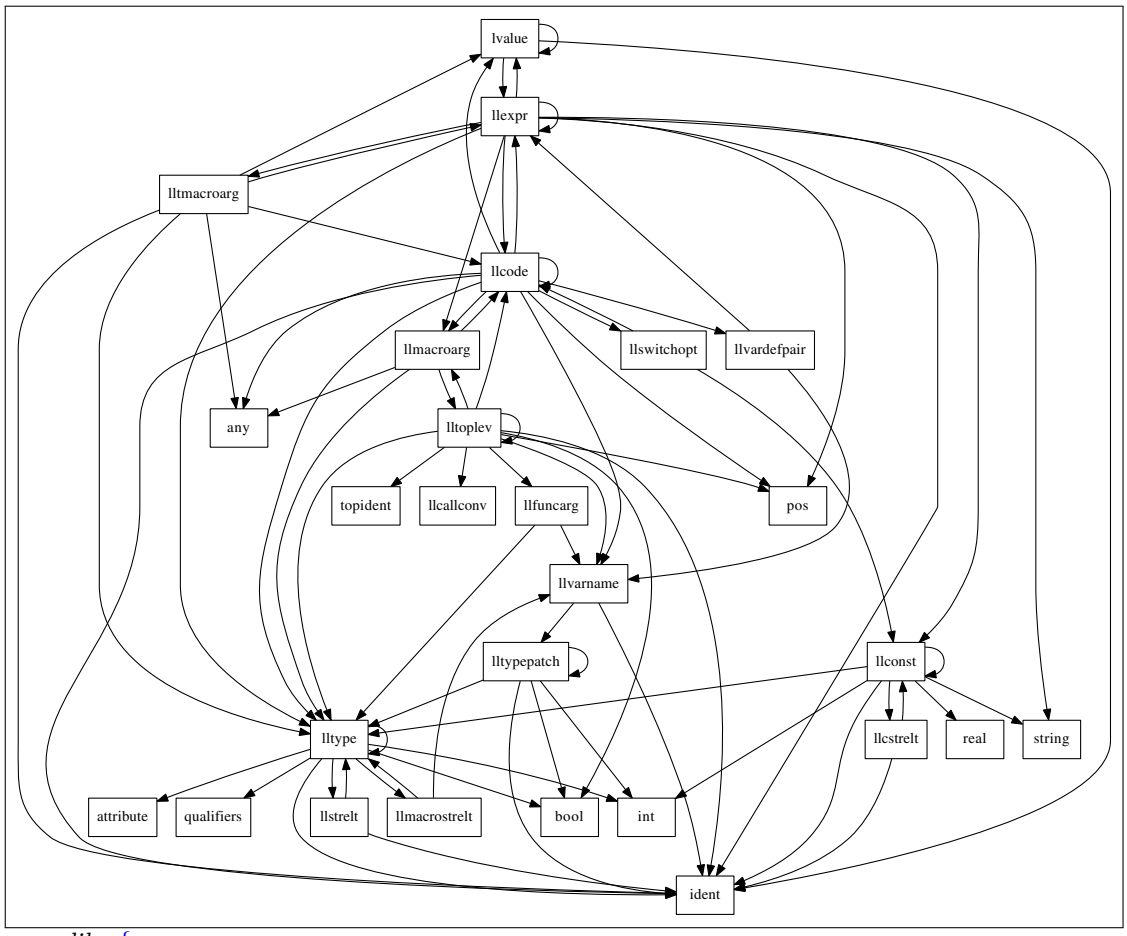
Meta Alternative Ltd.

2008-2011

Contents

1 ASTs definitions

1.1 The source language (generated by a parser)




```
ast clike {
  lltoplev =
      cfunc(pos:LOC, llcallconv:cc, lltype:ret, topident:name, bool:va,
            *llfuncarg:args, llcode:body)
    | efunc(pos:LOC, llcallconv:cc, lltype:ret, topident:name, bool:va,
            *llfuncarg:args)
    | typedef(pos:LOC, lltype:tp, topident:name)
    | global(pos:LOC, lltype:tp, llvarname:name)
    | eglobal(pos:LOC, lltype:tp, llvarname:name)
    | begin(.*lltoplev:es)

    // Should not appear after macro expansions
    | macroapp(ident:nm, .*llmacroarg:args)
```

```
  // For storing intermediate LLVM code:
  | xfunc(lltype:ret, topident:name, bool:va, . *llfuncarg:args)
  | xglobal(lltype:tp, topident:name)
  ;

topident is ident:v; // a dummy node

llmacroarg =
    stmt(llcode:s)
  | top(lltoplev:t)
  | type(lltype:t)
  | verb(any:v)
  ;

lltmacroarg =
    stmt(llcode:s)
  | expr(llexpr:s)
  | lvalue(lvalue:s)
  | var(ident:v)
  | type(lltype:t)
  | verb(any:v)
  ;

llvarname = v(ident:name)
          // Should not be present after macro expansion:
          | p(lltypepatch:p)
          ;

llfuncarg is (lltype:tp, llvarname:name);

llcode =
    begin(. *llcode:es)
  | label(pos:LOC, ident:lbl)
  | vardef(lltype:tp, llvarname:name)
  | set(pos:LOC, lvalue:l, llexpr:e)
  | expr(llexpr:e)
  | return(pos:LOC,llexpr:e)
  | vreturn(pos:LOC)
  | goto(pos:LOC, ident:lbl)
  | for(pos:LOC, *llcode:init, llexpr:cnd, llcode:step, llcode:body)
  | while(pos:LOC, llexpr:cnd, llcode:body)
  | do(pos:LOC, llcode:body, llexpr:cnd)
  | switch(pos:LOC, llexpr:e, *llswitchopt:opts, *llcode:dflt)
  | if3(pos:LOC, llexpr:e, llcode:tr, llcode:fl)
  | if2(pos:LOC, llexpr:e, llcode:tr)
```

```
  | break(pos:LOC)
  | nop()

  // Valid till the type propagation pass only
  | varinit(ident:l, llexpr:r)

  // Top level things lifting, should be eliminated right after typing
  //    pass (as it may be a result of a macro application)
  | toplift(any:t)

  // A temporary node for clike2 translation only, should never be generated
  | passexpr(llexpr:e)
  | passlvalue(lvalue:e)

  // Should not appear after macro expansions
  | macroapp(ident:nm, .*llmacroarg:args)
  | manyvardefs(lltype:tp, .*llvardefpair:vars)
  | protofor(pos:LOC, *llcode:init, *llexpr:cnds, *llexpr:steps, llcode:body)
  ;

llvardefpair = s(llvarname:nm)
             | d(llvarname:nm, llexpr:ini)
             ;

llswitchopt is (llconst: value, llcode:action);

llexpr =
    inblock(pos:LOC, llcode:c, llexpr:r)
  | call(pos:LOC, ident:id, .*llexpr:args)
  | callptr(pos:LOC, lvalue:fn, .*llexpr:args)
  | stdcallpfx(llexpr:e)
  | bin(pos:LOC, ident:op, llexpr:l, llexpr:r)

  | compop(pos:LOC, ident:op, llexpr:l, llexpr:r)
  | tri(llexpr:cnd, llexpr:tr, llexpr:fl)

  | un(ident:op, llexpr:e)
  | typecast(lltype:t, llexpr:e)
  | pre(ident:op, lvalue:v, .*lltype:vtyp)
  | post(ident:op, lvalue:v, .*lltype:vtyp)

  | modop(pos:LOC, ident:op, lvalue:l, llexpr:r)

  | eset(pos:LOC, lvalue:v, llexpr:e)
  | const(llconst:c)
  | globstring(string:s)
```

```
  | logand(pos:LOC, .*llexpr:es)
  | logor(pos:LOC, .*llexpr:es)

  | var(ident:nm)
  | arg(ident:nm)
  | glob(ident:nm)
  | globfun(ident:nm)

  | array(lvalue:ar, .*llexpr:idxs)
  | ref(lvalue:e)
  | deref(llexpr:e)
  | getelt(lvalue:e, ident:fldnm)
  | sizeof(lltype:t)

  // Special dual-stage macro expansion (propagation + post-propagation)
  | typedmacro(ident:nm, .*lltmacroarg:args)

  // Should not appear after macro expansions
  | macroapp(ident:nm, .*llmacroarg:args)
  | protoinblock(pos:LOC, .*llcode:c)
  | commaexprs(pos:LOC, .*llexpr:es)
  ;

lvalue =
    var(ident:nm)
  | glob(ident:nm)
  | globfun(ident:nm)
  | arg(ident:nm)
  | array(lvalue:ar, .*llexpr:idxs)
  | deref(llexpr:e)
  | getelt(lvalue:e, ident:fldnm)
  ;

llconst =
    null()
  | integer(ident:itype, int:v)
  | real(ident:rtype, real:v)
  | string(string:s)
  | constarray(. *llconst:elts)
  | conststruct(lltype:t, .*llcstrelt:elts)
  | zero(lltype:t)
  ;

llcstrelt is (ident:fld, llconst:v);
llstrelt is (ident:fld, lltype:t);
```

```
  llmacrostrelt is (lltype:t, llvarname:fld);

  lltypepatch =
    | ptr(lltypepatch:t)
    | fun(lltypepatch:ret, bool:va, .*lltype:args)
    | array(lltypepatch:t, .*int:dims)
    | v(ident:nm)
    ;

  lltype =
      integer(ident:itype)
    | real(ident:rtype)
    | alias(ident:x)
    | struct(*ident:nm, .*llstrelt:ts)
    | structalias(ident:nm)
    | structref(ident:nm)
    | ptr(lltype:t)
    | fun(lltype:ret, bool:va, .*lltype:args)
    | array(lltype:t, .*int:dims)
    | string()
    | void()

    // Qualified type
    | qual(qualifiers:c, lltype:t)
    | attr(attribute:a, lltype:t) // source level, to be translated to qual

    // Invalid after macro expansion
    | macrostruct(*ident:nm, .*llmacrostrelt:ts)

    // Intermediate, used for transforms
    | null()
    | bool()
    | arg(lltype:t)
    | nop()
    ;
}
```

1.2 Same language with all the expressions annotated with their types

```
ast clike2 : clike ( llexpr ↦ lloexpr, lvalue ↦ olvalue,
                     llvarname ↦ ollvarname ) {
  llexpr is (lltype:t , . lloexpr:e);
  lvalue is (lltype:t , . olvalue:e);
}
```

1.3 A first intermediate language

This intermediate language is already mostly LLVM, but expressions are allowed to be nested and types annotations are still present.

```
ast clike3 {
  llstmt2 =
      set(ident:nm, llexpr2:e)
    | setstring(ident:nm, string:s)
    | ret(llexpr2:value)
    | vret()
    | br(llexpr2:cnd, irlabel:tr, irlabel:fl)
    | br_label(ident:nm)
    | switch(llexpr2:value, irlabel:els, *irswitchdst:cases)
    | store(llexpr1:ptr, llexpr2:e)
    | storevar(ident:ptr, llexpr2:e)
    | label(ident:nm)
    | begin(.*llstmt2:es)
    | nop()
    // An intermediate instruction, must be removed before compilation
    | break()
    ;

  irswitchdst is ( llval:value, irlabel:dst );

  llexpr2 is (lltype:t,.llexpr1:e);

  llexpr1 =
     binary(irbinop:op, llexpr2:l, llexpr2:r)
   | extractelement(int:n, llexpr2:v, llexpr2:idx)
   | insertelement(int:n, irtype:t, llexpr2:v, llexpr2:elt, llexpr2:idx)
   | shufflevector(int:n1, llexpr2:val1, int:n1, llexpr2:val2, llexpr2:mask)
   | extractvalue(iraggtype:t, llexpr2:v, llexpr2:idx)
   | insertvalue(llexpr2:v, irtype:tv, llexpr2:elt, llexpr2:idx)
   | alloca(irtype:t)
   | load(llexpr2:ptr)
   | loadvar(ident:id) // shortcut
   | getelementptr(llexpr2:ptr, . *llexpr2:idxs)
   | getelementptr_inbounds(llexpr2:ptr, . *llexpr2:idxs)
   | convop(irconvop:op, llexpr2:v, irtype:t)
   | icmp(iricond:vcond, llexpr2:lhs, llexpr2:rhs)
   | fcmp(irfcond:vcond, llexpr2:lhs, llexpr2:rhs)
   | phi(irtype:t, .*irphi:dsts)
   | select(llexpr2:vif, llexpr2:vthen, llexpr2:velse)
   | call(ident:fn, .*llexpr2:args)
```

```
  | callptr(llexpr2:fn, .*llexpr2:args)
  | callptrstd(llexpr2:fn, .*llexpr2:args)
  | ptr(llexpr2:src, irtype:dst)
  | liftstatements(llstmt2:s, llexpr2:e)
  | val(llval:v)
  | stringtmp(string:s)
  ;

 llval =
        false()
      | true()
      | null(irtype:t)
      | integer(int:v, .*ident:itp)
      | float(float:v, .*ident:ftp)
      | struct(ident:nm, . *irstructel:elts)
      | array(irtype:t, . *llval:elts)
      | vector(. *llval:elts)
      | zero(irtype:t)
      | undef(irtype:t)
      | string(string:s)
      | blockaddress(irfunction:fn, irblock:blk)
      | var(ident:nm)
      | global(ident:nm)
      | globalfun(ident:nm)
      | sizeof(irtype:t)
      ;
}
```

2 Parser

This is a default clike parser, which produces an initial AST.
It is in many ways different from the original C language — it is designed with extensibility in mind, and it contains a number of additional keywords for marking specific parsing entries inside a quasiquotation syntax.

```
parser pfclike ( pfront ) {
```

Main parser entry

```
  pfclike ⇐ [cltop]:x [Spaces]* ⇒ x;
```

A global ignorance rule: omit all the occurences of the “Spaces” regular expression in front of all the tokens (lexical, keyword and named)

```
  !!Spaces;
```

Standard tokens rules, useful for syntax highlighting

```
[lexical:] <= [lexical] => {ctoken = lexic};
[keyword:] <= [keyword] ![IdentRest] => {ctoken = keyword};[1]
```

Floating point numbers regular expression

```
@@rcldouble <= ("-"/"+")? [Digit]+
         (((".") [Digit]+)?
            ("e"/"E") ("-"/"+")? [Digit]+)
         / ("." [Digit]+)
            );

@tkcldouble <= [rcldouble] => {ctoken = const};

cldouble <= [tkcldouble]:v => $sval(v);
```

Keywords that should not be parsed as identifiers; Some keywords are clike–specific, some came from C.

```
@@clkeywordI <= "break"/"null"/"NULL"/"case"/"switch"/"if"/"else"/"for"/"do"/
            "switch"/"type"/"code"/"return"/"default"/
            "expr"/"top"/"typedef"/"extern"/"inblock"/
            "void"/"float"/"double"/"struct"/"sizeof"/"var"/"lift"/
      "__stdcall"/"__llvm"
      ;
```

A regular expression for identifiers

```
@@clidenti <= ("_"/[Letter]) [IdentRest]*;

@@clkeyword <= [clkeywordI] ![IdentRest];

@clidenttk <= ![clkeyword] [clidenti];

clident <= [clidenttk]:v => {ctoken=ident} $sval(v);

clqident <= { "\" [clident]:i "\" => {qstate = unquote} unquote(i) }
       / [clident]
       ;
```

[1] Avoid parsing `ifabc` as `if` + identifier `abc`

A regular expression for Char literals

```
@clchart ⇐ [QUOTE] . [QUOTE] ⇒ {ctoken=lexic};
clchar ⇐ [clchart]:c ⇒ $charcode($stripval(c));
```

Top–level entries (typedefs, function definitions and declarations, etc.)

```
cltopatom ⇐ [cltop_start]
      / {typedef [cltype]:t [clqident]:nm ";" ⇒
           {mode=top} typedef($source(), t, nm) }
      / {extern? [clfuncsignature]:sig ";" ⇒
           {mode=top} efunc($source(), @sig) }
      / {extern? [clcleanfuncsignature]:sig ";" ⇒ sig }
      / {[clfuncsignature]:sig "{" eslist<[clcode]>:es "}" ⇒
           {mode=top} cfunc($source(), @sig,begin(@es)) }
      / {extern [clglob]:g ";" ⇒ {mode=top} eglobal($source(), @g) }
      / {[clglob]:g ";" ⇒ {mode=top} global($source(), @g) }
      / [cltop_inner]
      / {"{" eslist<[cltop]>:x "}" ";"? ⇒ begin(@x) }
      / { top "\" [clident]:i "\" ";" ⇒ {qstate = unquote} unquote(i) }
  ;
```

Compound top–level entries

```
cltop ⇐ [cltopatom]
      / {"#" [clqident]:nm "(" ecslist<[clmcarg],",">:args ")" ";" ⇒
         macroapp(nm,@args) }
```

The following rule needs a more elaborate explanation. Here and throughout the rest of the parser we're using parser entry keywords (like `top`) to specify entries we're substitutiong inside a quasiquotation syntax.

```
      / { ";" ⇒ begin() }
      ;
```

A helper term for detecting vararg function declarations

```
clfuncsigva ⇐ { "," "..." ⇒ 'true }
          / { !"," ⇒ $nil() }
       ;
```

[2] This is one of the extension points

[3] One of the notable differences from C — toplevel statements can be grouped inside curly braces.

[4] A clike macro application

Some calling conventions attributes, but we're not going to support the whole list here.

```
clcallconv ⇐ {"__stdcall" ⇒ stdcall()}
           / {"__llvm" ⇒ llvm()}
         ;
```

A function type parser

```
clfuncsignature ⇐ [clcallconv]:cc? [cltype]:t [clqident]:name "("
                    ecslist<[clsigarg],",">:args
                    [clfuncsigva]:va ")" ⇒
                  $cdr(f(cc,t,name,va,args));

clcleanfuncsignature ⇐ [clcallconv]:cc?  [cltype]:t [clqident]:name "("
                    ecslist<[cltype1],",">:args [clfuncsigva]:va
                    ")" ⇒ {mode=top} efunc($source(), cc,t,name,va,args);

cltype1 ⇐ [cltype]:t ⇒ $list(t,v(‘non));

clsigarg ⇐ [cltypebase]:t [clvarname]:name ⇒ $cdr(g(t,name));

clglob ⇐ [cltypebase]:t [clvarname]:name ⇒ $cdr(g(t,name));
```

A macro argument parser, it is not quite C–ish

```
clmcarg ⇐
        { type [cltype]:t ⇒ type(t) }
      / { code [clcode]:c ⇒ stmt(c) }
      / { expr [clexpr]:e ⇒ stmt(expr(e)) }
      / { top [cltop]:t ⇒ top(t) }
      / { "=pf" ":" [atopexpr]:e ⇒ verb(e) } /* fall back to pfront */
      / { "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
      / { [cltop]:t ⇒ top(t) }
      / { [clcode]:c ⇒ stmt(c) }
      / { [clexpr0]:e ⇒ stmt(expr(e)) }
      ;
```

Data types declarations

```
cltype ⇐
        { [cltypeattr]:ta [cltype]:t ⇒ attr(ta,t) }
      / { [cltype]:t "*" ⇒ ptr(t) }
      / { [cltype]:t "(" ecslist<[cltype],",">:args ")" ⇒ fun(t, $nil(),
```

```
@args) }
      / { [cltype]:t "[" "]" ⇒ ptr(t) }
      / { [cltype]:t "[" [number]:n "]" ⇒ array(t,n) }
      / [cltypebase]
      ;

cltypeattr ⇐
        { addrspace [number]:n ⇒ addrspace(n) }
      / { const ⇒ a(‘const) }
      ;

clvarname ⇐ [clvarnamex]:v ⇒ p(v);

clvarnamex ⇐
        { [clvarnamex]:t "[" "]" ⇒ ptr(t) }
      / { [clvarnamex]:t "[" [number]:n "]" ⇒ array(t,n) }
      / { "(" "*" [clvarnamex]:t ")" "(" ecslist<[cltype],","><:args ")" ⇒
          ptr(fun(t,$nil(),@args)) }[5]
      / [clvarnameatom];

clvarnameatom ⇐
        { "*" [clvarnamex]:t ⇒ ptr(t) }
      / { [clqident]:nm ⇒ v(nm) }
      ;

cltypebase ⇐
        { "::type" "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
      / { "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
      / [cltype_start]
      / { "int32" ⇒ integer(‘i32) }
      / { "int8"  ⇒ integer(‘i8)  }
      / { "int16"  ⇒ integer(‘i16)  }
      / { "int64"  ⇒ integer(‘i64)  }

      / { "uint32"  ⇒ integer(‘u32)  }
      / { "uint8"  ⇒ integer(‘u8)  }
      / { "uint16"  ⇒ integer(‘u16)  }
      / { "uint64"  ⇒ integer(‘u64)  }

      / { "void"   ⇒ void() }
      / { "float"   ⇒ real(‘float) }
      / { "double"   ⇒ real(‘double) }
      / { struct [clqident]:selfname? "{" slist<[clstrelt]>:elts "}" ⇒
            macrostruct(selfname,@elts) }
```

[5] A special case: function pointer application

```
    / { struct [clqident]:selfname ⇒
            structalias(selfname) }
    / [cltype_inner]
    / { [clqident]:id ⇒ alias(id) }
    ;

clstrelt ⇐ { [cltypebase]:t [clvarname]:nm ";" ⇒ $list(t,nm) }
    ;

clivpair ⇐ { [clvarname]:nm "=" [clexpr0]:e ⇒ d(nm,e) }
    / { [clvarname]:nm ⇒ s(nm) }
    ;

clswitchelt ⇐ case [clconst]:c ":" eslist<[clcode]>:bd ⇒
            $list(c, begin(@bd));

clsdefault ⇐ default ":" eslist<[clcode]>:bd ⇒ begin(@bd);
```

Function body: statements

```
clcode ⇐
      { "{" eslist<[clcode]>:es "}" ⇒ begin(@es) }
    / [clcode_start]
    / { lift "{" [cltop]:t ";"? "}" ⇒ toplift(t) }
    / { if "(" [clexpr]:cnd ")" [clcode]:tr else [clcode]:fl
        ⇒ {mode=stmt} if3($source(), cnd, tr, fl) }
    / { if "(" [clexpr]:cnd ")" [clcode]:tr ⇒
        {mode=stmt} if2($source(), cnd, tr) }
    / { while "(" [clexpr]:cnd ")" [clcode]:body ⇒
        {mode=stmt} while($source(), cnd, body) }
    / { do [clcode]:body while "(" [clexpr]:cnd ")" ⇒
        {mode=stmt} do($source(), body, cnd) }
    / { for "(" ([clfor1]:f1)? ";" ecslist<[clexpr0],",">:f2 ";"
            ecslist<[clexpr0],",">:f3 ")" [clcode]:body ⇒
          {mode=stmt} protofor($source(), f1,f2,f3,body) }
    / {"#" [clqident]:nm "(" ecslist<[clmcarg],",">:args ")" ";" ⇒
        {mode=stmt} macroapp($source(), nm,@args) }
    / { goto [clqident]:id ";" ⇒ {mode=stmt} goto($source(),id) }
    / { break ";" ⇒ {mode=stmt} break($source()) }
    / { return [clexpr]:e ";" ⇒ {mode=stmt} return($source(),e) }
    / { return ";" ⇒ {mode=stmt} vreturn($source()) }
    / { switch "(" [clexpr]:e ")" "{" slist<[clswitchelt]>:elts
        [clsdefault]:dflt? "}" ⇒
          {mode=stmt} switch($source(), e,elts,dflt) }
    / [clcode_inner]
```

```
      / { [clqident]:id ":" ⇒ {mode=stmt} label($source(), id) }
      / { [cltypebase]:tp cslist<[clivpair],",">:vars ";" ⇒
          manyvardefs(tp,@vars) }
      / { var [clqident]:l "=" [clexpr]:r ";" ⇒ varinit(l,r) }
      / { [cllvalue]:l "=" [clexpr]:r ";" ⇒
          {mode=stmt} set($source(),l,r) }
      / { "::code" "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
      / { "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
      / { [clexpr]:e ";" ⇒ expr(e) }
      / { ";" ⇒ begin() }
      ;

clfor1 ⇐ {[clfor0]:a "," cslist<[clfor0],",">:b ⇒ begin(a,@b) }
      / [clfor0]
      ;

clfor0 ⇐ { [cltypebase]:tp cslist<[clivpair],",">:vars ⇒
           manyvardefs(tp,@vars) }
      / { [clexpr0]:e ⇒ expr(e) }
      ;
```

Function body: expressions

```
clexpr ⇐ {"::comma" [clexpr0]:a "," cslist<[clexpr0],",">:b ⇒
         {mode=expr} commaexprs($source(), a,@b) }
      / [clexpr0]
      ;

clexpr0 ⇐ [classexpr] / [cltriexpr] ;

classexpr ⇐
          { [cllvalue]:l "=" [clexpr]:r ⇒
        {mode=expr} eset($source(),l,r) }[6]
       / { [cllvalue]:l "+=" [clexpr]:r ⇒
        {mode=expr} modop($source(),‘add,l,r) }
       / { [cllvalue]:l "-=" [clexpr]:r ⇒
        {mode=expr} modop($source(),‘sub,l,r) }
       / { [cllvalue]:l "*=" [clexpr]:r ⇒
        {mode=expr} modop($source(),‘mul,l,r) }
       / { [cllvalue]:l "/=" [clexpr]:r ⇒
        {mode=expr} modop($source(),‘div,l,r) }
       / { [cllvalue]:l "%=" [clexpr]:r ⇒
        {mode=expr} modop($source(),‘rem,l,r) }
       / { [cllvalue]:l "<<=" [clexpr]:r ⇒
```

[6] Assignment is an expression rather than a statement, as it yields a value in C

```
        {mode=expr} modop($source(),`shl,l,r) }
      / { [cllvalue]:l ">>=" [clexpr]:r ⇒
        {mode=expr} modop($source(),`shr,l,r) }
      / { [cllvalue]:l "&&=" [clexpr]:r ⇒
        {mode=expr} modop($source(),`logand,l,r) }
      / { [cllvalue]:l "||=" [clexpr]:r ⇒
        {mode=expr} modop($source(),`logor,l,r) }
      / { [cllvalue]:l "&=" [clexpr]:r ⇒
        {mode=expr} modop($source(),`and,l,r) }
      / { [cllvalue]:l "|=" [clexpr]:r ⇒
        {mode=expr} modop($source(),`or,l,r) }
      / { [cllvalue]:l "^=" [clexpr]:r ⇒
        {mode=expr} modop($source(),`xor,l,r) }
      / { [classexpr_inner]:a ⇒ a }
      ;

cltriexpr ⇐ { [clbinexpr]:cnd "?" [clexpr]:tr ":" [clexpr]:fl ⇒
            tri ... }[7]
      / [clbinexpr]
      ;

binary clbinexpr ⇐
        (100) [clbinexpr] "||" [clbinexpr] ⇒
            {mode=expr} logor($source(), L,R)
      | (100) [clbinexpr] "|" [clbinexpr] ⇒
          {mode=expr} bin($source(),`or,L,R)
      | (100) [clbinexpr] "^" [clbinexpr] ⇒
          {mode=expr} bin($source(),`xor,L,R)

      | (200) [clbinexpr] "&&" [clbinexpr] ⇒
            {mode=expr} logand($source(), L,R)
      | (200) [clbinexpr] "&" [clbinexpr] ⇒
          {mode=expr} bin($source(),`and,L,R)

      | (300) [clbinexpr] "==" [clbinexpr] ⇒
            {mode=expr} compop($source(),`eq,L,R)
      | (300) [clbinexpr] "!=" [clbinexpr] ⇒
      {mode=expr} compop($source(),`ne,L,R)

      | (600) [clbinexpr] "*" [clbinexpr] ⇒
          {mode=expr} bin($source(),`mul,L,R)
      | (600) [clbinexpr] "/" [clbinexpr] ⇒
        {mode=expr} bin($source(),`div,L,R)
      | (600) [clbinexpr] "%" [clbinexpr] ⇒
```

[7] A ternary expression

```
        {mode=expr} bin($source(),`rem,L,R)
      | (600) [clbinexpr] "<<" [clbinexpr] ⇒
        {mode=expr} bin($source(),`shl,L,R)
      | (600) [clbinexpr] ">>" [clbinexpr] ⇒
        {mode=expr} bin($source(),`shr,L,R)


      | (400) [clbinexpr] "<=" [clbinexpr] ⇒
          {mode=expr} compop($source(),`le,L,R)
      | (400) [clbinexpr] "<" [clbinexpr] ⇒
        {mode=expr} compop($source(),`lt,L,R)
      | (400) [clbinexpr] ">=" [clbinexpr] ⇒
        {mode=expr} compop($source(),`ge,L,R)
      | (400) [clbinexpr] ">" [clbinexpr] ⇒
        {mode=expr} compop($source(),`gt,L,R)

      | (500) [clbinexpr] "+" [clbinexpr] ⇒
          {mode=expr} bin($source(),`add,L,R)
      | (500) [clbinexpr] "-" [clbinexpr] ⇒
        {mode=expr} bin($source(),`sub,L,R)

      | [clunexpr]
    ;

clunexpr ⇐ { "-" [clprimexpr]:p ⇒ un(`minus,p) }
       / { "++" [cllvalue]:p ⇒ pre(`inc,p) }
       / { "--" [cllvalue]:p ⇒ pre(`dec,p) }
       / { "!" [clprimexpr]:p ⇒ un(`not,p) }
       / [clunexpr_inner]
       / [clpostexpr]
       ;

clpostexpr ⇐ { [cllvalue]:e "++" ⇒ post(`inc, e) }
       /  { [cllvalue]:e "--" ⇒ post(`dec, e) }
       /  [clpostexpr_inner]
       /  [clprimexpr]
       ;
```

A left–recursive expression syntax core:

```
clprimexpr ⇐
          { [clprimexpr]:e "[" [clexpr]:idx "]" ⇒ array(e,idx) }
       /  { [clprimexpr]:e "." [clqident]:fld ⇒ getelt(e, fld) }
       /  { [clprimexpr]:e "->" [clqident]:fld ⇒ getelt(deref(e),fld) }
    /  [clprimexpr_inner]
```

```
        /  [clprimexpratom]
        ;

clprimexpratom ⇐
            { "__stdcall" [clqident]:fn "(" ecslist<[clexpr0],","> :args ")" ⇒
              {mode=expr} stdcallpfx(call($source(),fn,@args)) }
        /   { "__stdcall" [cllvalue]:fn "(" ecslist<[clexpr0],","> :args ")"
               ⇒ {mode=expr} stdcallpfx(callptr($source(), fn,@args)) }
        /   { [clqident]:fn "(" ecslist<[clexpr0],","> :args ")" ⇒
               {mode=expr} call($source(), fn,@args) }
        /   { [cllvalue]:fn "(" ecslist<[clexpr0],","> :args ")"
               ⇒ {mode=expr} callptr($source(), fn,@args) }
        /   [clprimexpratom_inner]
        /   [clatomexpr]
        ;
```

And a very similar l–value left–recursive syntax: we want to destinguish l–values from all the other epxressions on a syntax level

```
cllvalue ⇐
            { [cllvalue]:e "[" [clexpr]:idx "]" ⇒ array(e,idx) }
        /   { [cllvalue]:e "." [clqident]:fld ⇒ getelt(e, fld) }
        /   { [cllvalue]:e "->" [clqident]:fld ⇒ getelt(deref(e),fld) }
        /   [cllvalueatom]
        ;

cllvalueatom ⇐   { "(" [cllvalue]:v ")" ⇒ v }
        /   { "*" [clatomexpr]:e ⇒ deref(e) }
        /   { "::lvalue" "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
        /   { "::lvar" "\" [clident]:i "\" ⇒ {qstate = unquote} var(unquote(i)) }
        /   { [clqident]:id ⇒ var(id) }
        ;
```

Atom expressions — please note that `::expr` entry is defined here, not at the expression entry node

```
clatomexpr ⇐
            { "(" [cltype]:t ")" [clexpr]:e ⇒ typecast(t,e) }
        /   { "(" [clexpr]:e ")" ⇒ e }
        /   { inblock "{" eslist<[clcode]>:es "}" ⇒
              {mode = expr} protoinblock($source(),@es) }
        /   { "&" [clatomexpr]:e ⇒ ref(e) }
        /   { "*" [clatomexpr]:e ⇒ deref(e) }
        /   { "#" [clqident]:nm "(" ecslist<[clmcarg],","> :args ")" ⇒
               macroapp(nm,@args) }
```

```
        /  { "::expr" "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
        /  { "\" [clident]:i "\" ⇒ {qstate = unquote} unquote(i) }
        /  [clexpr_inner]
        /  { [clconst]:c ⇒ const(c) }
        /  { sizeof "(" [cltype]:t ")" ⇒ sizeof(t) }
        /  { [clqident]:id ⇒ var(id) }
        /  { "::var" "\" [clident]:i "\" ⇒ {qstate = unquote} var(unquote(i)) }
        ;
```

Constant literals

```
clconst ⇐
          { [cldouble]:d ⇒ real(`double, d) }[8]
        / { [number]:n ⇒ integer(`i32, n) }
        / { [hexnumber]:n ⇒ integer(`i32, n) }
        / { [clchar]:c ⇒ integer(`i8, c) }
        / { [string]:s ⇒ string(s) }
        / { null ⇒ null() }
        / { NULL ⇒ null() }
        / [clconst_inner]
    / [clconstcompound]
        ;
```

Compound literals (no vectors yet, sorry)

```
clconstcompound ⇐
          { "[" cslist<[clconst],",">:cc ","? "]" ⇒ constarray(@cc) }
    / { [cltype]:tp "{" ecslist<[clconstfield],";">:flds ";"? "}" ⇒
              conststruct(tp,@flds) }
    ;
clconstfield ⇐
          [clqident]:nm "=" [clconst]:vl ⇒ $list(nm,vl);
```

Extension entry points:

```
&cltop_start; &cltop_inner; &cltype_start; &cltype_inner;
&clcode_start; &clcode_inner; &clprimexpr_inner; &clprimexpratom_inner;
&classexpr_inner;
&clunexpr_inner;
&clpostexpr_inner; &clexpr_inner; &clconst_inner;
}
```

[8] Doubles are stored in AST in their string form

3 Initial expansion pass

This is the first pass to be executed over a just parsed AST. Several different things are done on this level. Firstly, clike macro applications are partially expanded (but not the typed macros — they'll be expanded in a type propagation pass). Secondly, a number of the initial AST oddities (introduced entirely for a sake of parsing simplicity) are substituted with a cleaner code.
We're getting rid of the "string" type here (the one we've introduced for string literals), structs and function declarations are simplified, comma–blocks are expanded into a more fundamental form, variable declaration initialisers are separated from the declarations, and `for` is simplified from initial `protofor` nodes. Standalone expressions are also translated into phoney `sets`.

Applies a macro, if it is valid, and then re-enters into the macro expansion loop.

```
function clike_apply_macro(env, nm, args, reenter)
{
  mcenv = env /@ "  :macros";
  if(not(mcenv)) ccerror('CLIKE:MACRO-ENV-UNDEFINED'(nm));
  mc = mcenv /@ nm;
  if(not(mc)) ccerror('CLIKE:MACRO-UNDEFINED'(nm));
  reenter(env, mc(env,args));
}
```

Expand all the macros inside an expression

```
function clike_expand_macros_expr(env, tl)
  visit:clike(llexpr: tl) {
   deep llexpr {
      macroapp ↦ clike_apply_macro(env, nm, args, clike_expand_macros_expr)
    | else ↦ node
   };
   once llcode : ∀ clike_expand_macros_code(env, node);
  }
```

Expands all the macros inside a statement

```
function clike_expand_macros_code(env, tl)
  visit:clike(llcode: tl) {
   deep llcode {
      macroapp ↦ clike_apply_macro(env, nm, args, clike_expand_macros_code)
    | else ↦ node
   };
   once llexpr : ∀ clike_expand_macros_expr(env, node);
  }
```

A user–defined macros expansion pass. Should be performed right after the core macros expansion pass (which means that user-defined macros should not construct core macros).

```
function clike_expand_macros_top(env, tl)
  visit:clike(lltoplev: tl) {
   deep lltoplev {
      macroapp ↦ clike_apply_macro(env, nm, args, clike_expand_macros_top)
    | else ↦ node
   };
   once llcode : ∀ clike_expand_macros_code(env, node);
  }
```

Expand the simplified type definitions representation

```
function clike_patch_type(t, p)
{
  n = mkref(∅);
  t1 = visit:clike(lltypepatch: p) {
         deep lltypepatch {
            v ↦ { n:=nm; return t; }
          | else ↦ node
         }
       };
  return (t1 : 'v'(^n));
}
```

Some core macros are build into clike ast, but must be expanded into simpler constructions before compilation begins and even before the user-defined macro expansion pass. The reason for this simple core macros is in the simplicity of the parser.

```
function clike_expand_core(tl)
{
  if((^clike_debug_level)>1) println(#'(SRC: ,tl));
  visit:clike(lltoplev: tl) {
    // llvarname is a part of a simplified type definition
    once llvarname {
       v ↦ λ(t) {t : node}
     | p ↦ λ(t) clike_patch_type(t,p)
     | else ↦ ccerror('CLIKE:OOPS'(node))
    };
    // Top level global definitions are converted from a simplified form
```

```
deep lltoplev {
   global ↦  { <nt:nn> = name(tp); mk:node(tp=nt, name=nn) }
 | eglobal ↦ { <nt:nn> = name(tp); mk:node(tp=nt, name=nn) }
 | else ↦ node
};
// Same for structure elements
deep llmacrostrelt : { <nt:nn> = fld(t); [cadr(nn);nt] };
// And functions arguments
deep llfuncarg : { <nt:nn> = name(tp); [nt; nn] };
// There is no underlying string type, so it is expanded here.
// Simplified structure is converted into a normal one
deep lltype {
   string ↦ 'ptr'('integer'('i8'))
 | macrostruct ↦ 'struct'(nm,@ts)
 | else ↦ node
};
deep llexpr {
   // Simple parsed 'in-expression-block' is converted into a normal one
   protoinblock ↦ 'inblock'(LOC,'begin'(@cuttail(c)),{
                    match lasttail(c) with
                      ['expr'(e)] ↦ e
                    | else ↦ ccerror('CLIKE:INCORRECT-INBLOCK'(node))
                  })
   // Comma-delimited list of expressions is translated into an in-block
 | commaexprs ↦ 'inblock'(LOC,'begin'(@map e in cuttail(es) do
                                'expr'(e)),
                          car(lasttail(es)))
 | else ↦ node
};
deep llcode {
   // Compiler backend knows nothing about variable initialisers,
   // so here 'manyvardefs' is expanded into simpler constructions.
   manyvardefs ↦ 'begin'(
       @map append vars do {
         match vars with
           s(nm) ↦ {
             <nt:nn> = nm(tp);
             ['vardef'(nt,nn)]
           }
         | d(nm,ini) ↦ {
             <nt:nn> = nm(tp);
             ['vardef'(nt,nn);'set'(∅, 'var'(cadr(nn)), ini)]
           }
       }
   )
   // For, as it parsed, should be translated into a simpler form to
```

```
        // be compiled.
      | protofor ↦ 'for'(LOC, init, (match cnds with
                                       [one] ↦ one
                                     | one : many ↦ 'logand'(∅,one,@many)
                                     | ∅ ↦ 'const'('integer'('i32',1))),
                                    (match steps with
                                       [one] ↦ 'expr'(one)
                                     | else ↦
                                         'begin'(@map steps do 'expr'(steps))),
                                    body)
        // A single embedded set expression is translated into a statement,
        // just for a better readability of an intemediate code.
      | expr ↦ (
          match e with
            'eset     '(l,n,v) ↦ 'set'(l,n,v)
          | else ↦ node
        )
      | else ↦ node
    }
  }
}
```

A shortcut for defining Clike macros in the default macro environment

```
#(macro clike_defmacro (name args . body)
   `(hashput clike_default_mcenv ,(S<< name)
      (fun (env macro-body)
        (format macro-body ,args
          ,@body))))
```

A shortcut for defining Clike typed macros in the default macro environment

```
#(macro clike_deftexpander (name args . body)
   `(hashput clike_default_mcenv ,(S<< " :typexpander: " name)
      (fun (env return_type macro-body)
        (format macro-body ,args
          ,@body))))
```

A shortcut for defining Clike typing rules in the default macro environment

```
#(macro clike_deftrules (name args . body)
   `(hashput clike_default_mcenv ,(S<< " :typrules: " name)
      (fun (env macro-body)
```

```
    (format macro-body ,args
      ,@body))))
```

4 Utility functions for the typing pass

```
// Strip from qualifiers
function clike_unqualify(tp)
  visit:clike(lltype:tp) {deep lltype {
     qual ↦ t
   | else ↦ node
  }}

// Check if a type is a structure
function clike_isstruct(x)
  match x with
     'struct'([nm],@_) ↦ nm
   | 'structalias'(nm) ↦ nm
   | 'structref'(nm) ↦ nm
   | else ↦ ∅

// Convert a type into a canonical form
function clike_type_canonical(t)
  visit:clike(lltype: clike_unqualify(t)) {deep lltype {
     struct ↦ {if(nm) 'structalias'(car(nm)) else node}
   | array ↦ 'ptr'(t)
   | else ↦ node
  }}

// Get s string representation of a type
function clike_type_string(t)
  %to-string(clike_type_canonical(t))

// Check if two types are identical
function clike_type_iso(a, b)
 do loop(t1=a, t2=b) {
   match t1:t2 with
      'integer'(t1):'integer'(t2) ↦ %eqv?(t1,t2)
    | 'ptr'(a):'ptr'(b) ↦ loop(a,b)
    | 'ptr'(a):'array'(b,@r1) ↦ loop(a,b)
    | 'array'(a,@r):'ptr'(b) ↦ loop(a,b)
    | 'array'(a,@r):'array'(b,@r1) ↦ and(loop(a,b),iso(r,r1))
    | x:y ↦ {
```

```
      s1 = clike_isstruct(x);s2=clike_isstruct(y);
      if(and(s1,s2)) %eqv?(s1,s2) else iso(x,y)
    }}

// Check if an integer is of a signed kind
function clike_signed_int(itype)
 case itype {
   'i8'|'i16'|'i32'|'i64' ↦ true
  | 'u8'|'u16'|'u32'|'u64' ↦ nil
  | else ↦ ccerror('CLIKE:INCORRECT-INTEGER-SPEC'(itype))
 }

// Check if a type is signed, if this notion applies
function clike_signed(tp)
  visit:clike(lltype: tp) {once lltype {
      integer ↦ clike_signed_int(itype)
    | real ↦ true
    | ptr ↦ nil
    | array ↦ nil
    | string ↦ nil
    | else ↦ ccerror('CLIKE:INCORRECT-TYPE'(tp))
  }}

// A representation for C strings
define clike_string_type = 'ptr'('integer'('i8'));

// A helper function which detects a type of a given constant literal
function clike_const_type(c)
 visit:clike(llconst:c) {
   deep llconst {
      null ↦ 'null'()
    | integer ↦ 'integer'(itype)
    | real ↦ 'real'(rtype)
    | string ↦ clike_string_type
    | constarray ↦ 'array'(caar(elts)) // this is why it is deep
    | conststruct ↦ t
   }}

// A helper function which returns an array element type
function clike_array_elt_type(tp)
 match tp with
    ptr(array(t,@x)) ↦ t
  | ptr(ptr(t)) ↦ t
  | arg(array(t,@x)) ↦ t
  | arg(ptr(t)) ↦ t
  | else ↦ ccerror('CLIKE:ARRAY-TYPE'(tp))
```

```
// A helper function which makes a reference type for a given type
function clike_make_ref_type(tp)
 return 'ptr'(tp)

// A helper function which returns a type referenced by a given reference type
function clike_deref_type(tp)
  match tp with
    ptr(e) ↦ e
  | else ↦ ccerror('CLIKE:DEREF-TYPE'(tp))

function clike_getstruct(tp)
  match tp with
     ptr(e) ↦ e
  |  else ↦ tp

// A helper function which gives a type of a named structure field
function clike_fieldtype(tp, fldnm)
  match clike_getstruct(tp) with
     struct(nm,@elts) ↦ {
          v = filter(λ(x) %eqv?(car(x),fldnm), elts);
          if(v) cadr(car(v)) else
             ccerror('CLIKE:STRUCT-NO-SUCH-FIELD'(tp, fldnm))}
   | else ↦ ccerror('CLIKE:STRUCT-TYPE'(tp))

// Returns a number of a field
function clike_fieldnumber(tp, fldnm)
  match clike_getstruct(tp) with
     struct(nm,@elts) ↦
          do loop(es = elts, i = 0)
          {
             match es with
                [nm;tp]:rest ↦ {
                   if(%eqv?(nm, fldnm)) i
                   else loop(rest,i+1)
                }
              | else ↦ ccerror('CLIKE:STRUCT-NO-SUCH-FIELD'(tp, fldnm))
          }
   | else ↦ ccerror('CLIKE:STRUCT-TYPE'(tp))

// Make a typed node with a binary expression, give it a type of a first
//  argument.
function clike_binopsimple(LOC,op,l,r)
  return car(l):'bin'(LOC,op,l,r)

function clike_modopsimple(LOC,op,l,r)
```

```
  return clike_deref_type(car(l)):'modop'(LOC,op,l,r)

// Cast one type to another.
// In LLVM, array of a fixed size and a pointer are different
function clike_castto(t,n)
   match t:car(n) with
      ptr(t1):null() ↦ t:'const'('null'())
    | ptr(t1):array(t2,@_) ↦
        if(clike_type_iso(t1,t2))
          t:'ref'(t:'array'('ptr'(car(n)):cdr(n),
                    ['nop']:'const'('integer'('i32',0))))
        else return t:'typecast'(t,n)
    | else ↦ return t:'typecast'(t,n)

// Adjust an integer to the size of a pointer type
function clike_ptrarith(LOC,op, ptr, i)
{
  <itp:iv> = i;
  <ptp:p>  = ptr;
   if(not(%eqv?(op,'add'))) ccerror(#'(POINTER OP NOT SUPPORTED));
   'ref'(ptp:'array'('ptr'(ptp):p,i))
}

// Adjust an integer to the size of a pointer type
function clike_ptrarithMOD(LOC,op, ptr, i)
{
  <itp:iv> = i;
  <ptp:p>  = ptr;
   pitp = clike_deref_type(ptp);
   if(not(%eqv?(op,'add'))) ccerror(#'(POINTER OP NOT SUPPORTED));
   'eset'(∅, ptr, pitp:'ref'(clike_deref_type(pitp):'array'(ptr,i)))
}

function clike_rank(i)
 case i {
   'i8' ↦ 1 | 'u8' ↦ 2 | 'i16' ↦ 3 | 'u16' ↦ 4
  | 'i32' ↦ 5 | 'u32' ↦ 6 | 'i64' ↦ 7 | 'u64' ↦ 8
 }

// Calculate the binary operation type, inject implicit casts if needed
function clike_fix_binoptypes(LOC, op, l,r)
{
   tl = clike_unqualify(car(l));tr = clike_unqualify(car(r));
   match tl:tr with
      integer(i1):integer(i2) ↦
        if(%eqv?(i1,i2)) clike_binopsimple(LOC,op,l,r)
```

```
        else {
          if(%>=(clike_rank(i1),clike_rank(i2)))
            tl : 'bin'(LOC,op,l,clike_castto(tl,r))
          else
            tr : 'bin'(LOC,op,clike_castto(tr,l),r)
        }
    | integer(i1):real(r2) ↦ tr:'bin'(LOC,op,clike_castto(tr,l),r)
    | real(r1):integer(i2) ↦ tl:'bin'(LOC,op,l,clike_castto(tl,r))
    | real('float'):real('double') ↦ tr:'bin'(LOC,op,clike_castto(tr,l),r)
    | real('double'):real('float') ↦ tl:'bin'(LOC,op,l,clike_castto(tl,r))
    | ptr(t1):integer(i2) ↦ tl:clike_ptrarith(LOC,op,l,r)
    | integer(t1):ptr(t2) ↦ tr:clike_ptrarith(LOC,op,r,l)
    | else ↦ clike_binopsimple(LOC,op,l,r)
}

// Calculate the binary mod operation type, inject implicit casts if needed
function clike_fix_modoptypes(LOC, op, l,r)
{
   tl = clike_deref_type(clike_unqualify(car(l)));tr = clike_unqualify(car(r));
   match tl:tr with
      integer(i1):integer(i2) ↦ if(%eqv?(i1,i2)) clike_modopsimple(LOC,op,l,r)
        else tl : 'modop'(LOC, op, l, clike_castto(tl,r))
    | integer(i1):real(r2) ↦ tl:'modop'(LOC,op,l,clike_castto(tl,r))
    | real(r1):integer(i2) ↦ tl:'modop'(LOC,op,l,clike_castto(tl,r))
    | real('float'):real('double') ↦ tl:'bin'(LOC,op,l,clike_castto(tl,r))
    | real('double'):real('float') ↦ tl:'bin'(LOC,op,l,clike_castto(tl,r))
    | ptr(t1):integer(i2) ↦ tl:clike_ptrarithMOD(LOC,op,l,r)
    | integer(t1):ptr(t2) ↦ tr:clike_ptrarithMOD(LOC,op,r,l)
        //TODO: report error
    | else ↦ clike_modopsimple(LOC,op,l,r)
}


// Construct a comparision operation node
function clike_compopsimple(LOC,op,l,r)
  return 'compop'(LOC,op,l,r)

// Fix the comparision operation arguments, if needed
function clike_fix_compoptypes(LOC,op,l,r)
{
   tl = clike_unqualify(car(l));tr = clike_unqualify(car(r));
   match tl:tr with
      integer(i1):integer(i2) ↦
        if(%eqv?(i1,i2)) clike_compopsimple(LOC,op,l,r)
        else {
          cparses(s) = %S->N(%S<<(cdr(%symbol->list(s))));
```

```
        n1 = cparses(i1);n2=cparses(i2);
        if(n1>n2) 'compop'(LOC,op,l,clike_castto(tl,r))
          else 'compop'(LOC,op,clike_castto(tr,l),r)
      }
  | integer(i1):real(r2) ↦ 'compop'(LOC,op,clike_castto(tr,l),r)
  | real(r1):integer(i2) ↦ 'compop'(LOC,op,l,clike_castto(tl,r))
  | real('float'):real('double') ↦ 'compop'(LOC,op,clike_castto(tr,l),r)
  | real('double'):real('float') ↦ 'compop'(LOC,op,l,clike_castto(tl,r))
  | ptr(t1):integer(i2) ↦ 'compop'(LOC,op,l,r)
  | ptr(t1):null() ↦ 'compop'(LOC,op,l,'ptr'(t1):'const'('null'()))
  | null():ptr(t1) ↦ 'compop'(LOC,op,'ptr'(t1):'const'('null'()),r)
  | integer(t1):ptr(t2) ↦ 'compop'(LOC,op,l,r)
  | else ↦ clike_compopsimple(LOC,op,l,r)
}

// Inject a cast into a right side of a set operation, if needed
function clike_fix_settype(set,loc,l,r)
 {
  lt = clike_deref_type(clike_unqualify(car(l)));
  rt = car(r);
  if(clike_type_iso(lt,rt)) [set;loc;l;r]
    else
    match lt:rt with
      ptr(t1):null() ↦ [set;loc;l;'ptr'(t1):'const'('null'())]
    | else ↦ [set;loc;l;clike_castto(lt,r)]
 }

// Inject casts into function arguments, if needed
function clike_fix_funcall(call,LOC,fn, va, args, atps)
{
  %__lcut(l1,l2) = do loop(a=l1,b=l2) if(a) loop(cdr(a),cdr(b)) else b;
  [call;LOC;fn;@map az in zip(args,atps) do {
             <[a; tt]> = az; at=clike_unqualify(car(a));
              if(clike_type_iso(at,tt))
                   return a
                else return clike_castto(tt,a);
            };
   @if(va) %__lcut(atps,args) else ∅
  ]
}

// A shortcut for building a zero comparision operation
function clike_notzero(e0)
{
  <tp:e> = e0;
  'compop'(∅,'ne',e0,tp:'const'('zero'(tp)))
```

```
}

// Fix boolean expressions - compare to zero if a value is not a
//   boolean already
function clike_fix_bool(e)
  match clike_unqualify(car(e)) with
    bool() ↦ e
  | else ↦ 'bool'() : clike_notzero(e)


function clike_decay(tp)
  visit:clike(lltype: tp) {
    once lltype {
      array ↦ 'ptr'(t)
    | else ↦ node
    }
  }
```

4.1 A compilation pass: types propagation

N.B. —- a typed macros expansion step is performed within this pass as well.

```
function clike_types_inner (env, c, toploop)
{
  vars = mkhash();
  do loop(c0 = c)
   visit:clike(llcode: c0)
   {
     deep llvarname {
        v ↦ name
      | p ↦ ccerror('CLIKE:WRONG-PASS'(node))
     };
     deep llexpr {
        call ↦ {
           nid = clike_env_name_resolve(env, id);
           <tp:va:atps> = clike_env_funcretargtypes(env, nid);
           return tp : clike_fix_funcall('call', LOC, nid, va, args, atps);
        }
      | callptr ↦ {
           <tp:va:atps> = clike_funcptrtype(car(fn));
           return tp : clike_fix_funcall('callptr', LOC, fn, va, args, atps);
        }
      | stdcallpfx ↦ car(e):node
      | bin ↦ clike_fix_binoptypes(LOC, op,l,r)
      | compop ↦ 'bool'() : clike_fix_compoptypes(LOC, op ,l,r)
      | un ↦ case op { 'minus' ↦ car(e) : node
```

```
                | 'not' ↦ clike_fix_bool(e) }
| tri ↦ car(tr) : (mk:node(cnd=clike_fix_bool(cnd)))
| typecast ↦ {
    t1 = clike_env_unitype(env, t);
    clike_castto(t1, e)
  }
| pre ↦ car(v) : mk:node(vtyp = [car(v)])
| post ↦ car(v) : mk:node(vtyp = [car(v)])
| inblock ↦ car(r) : node
| eset ↦ clike_deref_type(car(v)) : clike_fix_settype('eset',LOC,v,e)
// TODO: implicit casts for modops
| modop ↦ clike_fix_modoptypes(LOC, op, l, r)
| globstring ↦ clike_string_type:node
| const ↦ {clike_const_type(c) :
          match c with
            ['string';s] ↦ 'globstring'(s)
          | else ↦ node }
| var ↦ {
   v1 = vars /@ nm;
   if(v1) v1:node else
   {
     v2 = clike_env_argtype(env, nm);
     if(v2) v2:'arg'(nm) else
     {
       nnm = clike_env_name_resolve(env, nm);
       v3 = clike_env_globtype(env, nnm);
       if(v3) v3:'glob'(nnm) else
       {  v4 = clike_env_globfunctype(env, nnm);
          if(v4) v4:'globfun'(nnm) else
            ccerror('CLIKE:UNKNOWN-VAR'(nm))
       }}}}
| arg ↦ { v2 = clike_env_argtype(env, nm); v2 : node }
| glob ↦ { v3 = clike_env_globtype(env, nm); v3 : node }
| globfun ↦ { v4 = clike_env_globfunctype(env, nm); v4:node }
| array ↦ clike_array_elt_type(car(ar)) : node
| ref ↦ car(e) : node // It's an lvalue already, must be a ref anyway
| deref ↦ clike_deref_type(car(e)) : node
| getelt ↦ clike_fieldtype(car(e), fldnm) : node
| sizeof ↦ 'integer'('i64') : node
| logand ↦ 'bool'():'logand'(LOC,@map es do clike_fix_bool(es))
| logor ↦ 'bool'():'logor'(LOC,@map es do clike_fix_bool(es))

// Applying type rules for a dual-stage macro:
| typedmacro ↦ {
    rule = clike_env_gettypingrules(env, nm);
    rtype = if(rule) rule(env, args) else ∅;
```

```
        expander = clike_env_gettypedexpander(env, nm);
        ncode_0 = expander(env, rtype, args); // args are going to be
            // stripped from types during this expansion
        ncode = loop('passexpr'(ncode_0)); // redo the propagation
        return ncode
      }
    | else ↦ ccerror('CLIKE:NOT-ALLOWED-HERE'(node))
};
deep llcode {
    vardef ↦ {
        ntp =  clike_env_unitype(env, tp);
        vars /! name <- clike_decay(ntp);
        cdr(env) /! name <- 'lvar'(ntp);
        return mk:node(tp = ntp);
      }
    | varinit ↦ {
        rtp = car(r);
        vars /! l <- clike_decay(rtp);
        cdr(env) /! l <- 'lvar'(rtp);
        return 'begin'('vardef'(rtp,l),
                       'set'(∅,'ptr'(rtp):'var'(l) , r))
      }
    | toplift ↦ {toploop(env, t); 'begin'()}
    | set ↦ clike_fix_settype('set',LOC,l,e)
    | passexpr ↦ return e
    | if2 ↦ mk:node(e=clike_fix_bool(e))
    | if3 ↦ mk:node(e=clike_fix_bool(e))
    | for ↦ mk:node(cnd=clike_fix_bool(cnd))
    | do ↦ mk:node(cnd=clike_fix_bool(cnd))
    | while ↦ mk:node(cnd=clike_fix_bool(cnd))
    | else ↦ return node
};
deep lvalue {
    var ↦ {
      <vt:vv> = loop('passexpr'(node));
      match vv with
          arg(_) ↦ 'arg'(vt):vv
        | else ↦ clike_make_ref_type(vt):vv
      }
  | array ↦ clike_make_ref_type(clike_array_elt_type(car(ar))):node
  | deref ↦ {
      match e with
          ptr(t):arg(a) ↦ e
        | array(t,@idxs):arg(a) ↦ e
        | t:x ↦ t:node
      }
```

```
        | getelt ↦ clike_make_ref_type(clike_fieldtype(car(e),fldnm)):node
        | else ↦ ccerror('CLIKE:WRONG-PASS'(node))
       };
    }
}
```

An additional tiny pass which replaces the abstract 'bool' with a concrete ingeger type. Bool was needed for fixing boolean expressions, and it should not interfere later with casting compilation.

```
function clike_clean_bools(code)
  visit:clike2(llcode: code) {
    deep lltype {
      bool ↦ 'integer'('i32')
     | else ↦ node
    }
  }
```

An interface function, binds all the typing passes together

```
function clike_types (env, c, toploop)
  clike_clean_bools(clike_types_inner(env,c,toploop))
```

Convert clike2 back into clike

```
function clike_untype_llcode(c)
  visit:clike2(llcode:c)
    { deep llcode {
        passexpr ↦ e
      | passlvalue ↦ e
      | else ↦ node
      };
      deep llexpr : e; deep lvalue : e;}
```

Convert clike2 back into clike

```
function clike_untype_llexpr(c)
   clike_untype_llcode('passexpr'(c))
```

Convert clike2 back into clike

```
function clike_untype_lvalue(c)
   clike_untype_llcode('passlvalue'(c))
```

5 Compiler

Compiles a CLike type into LLVM type

```
function clike_c_type(env, tp)
        clike_c_type0(env, tp, ∅)
```

An actual implementation:

```
function clike_c_type0(env, tp, arrp)
 visit:clike2(lltype: tp) {
  deep lltype {
    struct ↦ 'struct'((if(nm) car(nm) else ∅),@ts)
  | integer ↦ node
  | real ↦ 'float'(rtype)
  | alias ↦ clike_c_type(env, clike_env_unitype(env, node))
  | structref ↦ node
  | ptr ↦ ( match t with
              void() ↦ 'pointer'('integer'('i8'))
            | else ↦ 'pointer'(t) )
  | fun ↦ if(va) 'varfunction'(ret,@args) else 'function'(ret,@args)
  | array ↦ if(arrp) 'array'(dims, t) else 'pointer'(t)
  | void ↦ node
  | else ↦ ccerror('CLIKE:TODO'(node))
 };
 once llstrelt : clike_c_type0(env, t, true);
}
```

The same as above, to be used after an array decay pass

```
function clike_ca_type(env, tp)
        clike_c_type0(env, tp, true)
```

Compile a CLike constant literal into an LLVM constant

```
function clike_c_const(env, c, nt)
   visit:clike(llconst: c)
   { once llconst {
         integer ↦ 'val'('integer'(v, itype))
       | real ↦ 'val'('float'(v, rtype))
       | null ↦ 'val'('null'(clike_c_type(env, nt)))
       | zero ↦ 'val'('zero'(clike_c_type(env, t)))
       | constarray ↦ {
           et = clike_array_elt_type(nt);
```

```
            'val'('array'(
                clike_c_type(env, et),
                @map elts do cadr(clike_c_const(env, elts, et))))
            }
          | else ↦ ccerror('CLIKE:NOT-IMPLEMENTED-YET'(c))
        }}
```

Compile a type conversion into LLVM

```
function clike_convop_llvm(tto, tfrom)
   match tfrom:tto with
      integer(i1):integer(i2) ↦ {
        if(i1 === i2) ∅ else
        { cparses(s) = %S->N(%list->string(cdr(%symbol->list(s))));
          n1 = cparses(i1);n2=cparses(i2);
          if(n1>n2) 'Trunc' else 'ZExt' // TODO: do something with signs
        }}
    | real('float'):real('double') ↦ 'FPExt'
    | real('double'):real('float') ↦ 'FPTrunc'
    | real(r):integer(i) ↦ if(clike_signed(tto)) 'FPToSI'
                           else 'FPToUI'
    | integer(i):real(r) ↦ if(clike_signed(tfrom)) 'SIToFP'
                           else 'UIToFP'
    | ptr(p1):ptr(p2) ↦ if(clike_type_iso(p1,p2)) ∅ else 'BitCast'
    | array(t1,@_):ptr(t2) ↦ if(clike_type_iso(t1,t2)) ∅ else 'BitCast'
    | ptr(t1):array(t2,@_) ↦ if(clike_type_iso(t1,t2)) ∅ else 'BitCast'
    | else ↦ ccerror('CLIKE:UNSUPPORTED-CAST'(tfrom,tto))
```

Compile an unary operation into LLVM

```
function clike_c_unop(env, op, e)
 case op {
   'minus' ↦
     'binary'('Sub',car(e):'val'('zero'(clike_c_type(env, car(e)))),e)
  | 'not' ↦ 'icmp'('EQ',e,car(e):'val'('zero'(clike_c_type(env, car(e)))))
 }
```

Compiles a binary opeartion into LLVM

```
function clike_binop_llvm(op, tp, l, r)
{ fp = match tp with real(_) ↦ true | else ↦ nil;
  trap() = ccerror('CLIKE:WRONG-TYPE'(tp));
  llop = case op {
           'add' ↦ if(fp) 'FAdd' else 'Add'
```

```
        | 'sub' ↦ if(fp) 'FSub' else 'Sub'
        | 'mul' ↦ if(fp) 'FMul' else 'Mul'
        | 'div' ↦ if(fp) 'FDiv' else {
                    if(clike_signed(tp)) 'SDiv' else 'UDiv'
                  }
        | 'rem' ↦ if(fp) 'FRem' else {
                    if(clike_signed(tp)) 'SRem' else 'URem'
                  }
        | 'shl' ↦ if(fp) trap() else 'Shl'
        | 'shr' ↦ if(fp) trap() else {if(clike_signed(tp)) 'AShr' else 'LShr'}
        | 'and' ↦ if(fp) trap() else 'And'
        | 'or' ↦ if(fp) trap() else 'Or'
        | 'xor' ↦ if(fp) trap() else 'Xor'
       };
  return 'binary'(llop, l, r)}
```

Compile a comarison operation

```
function clike_compop_llvm(op, tp, l, r)
{
  fp = match tp with real(_) ↦ true | else ↦ nil;
  sig = clike_signed(tp);
  llop = if(not(fp))
         case op {
           'eq' ↦ 'EQ'
         | 'ne' ↦ 'NE'
         | 'gt' ↦ if(sig) 'SGT' else 'UGT'
         | 'ge' ↦ if(sig) 'SGE' else 'UGE'
         | 'lt' ↦ if(sig) 'SLT' else 'ULT'
         | 'le' ↦ if(sig) 'SLE' else 'ULE'
         } else case op {
           'eq' ↦ 'OEQ'
         | 'ne' ↦ 'ONE'
         | 'gt' ↦ 'OGT'
         | 'ge' ↦ 'OGE'
         | 'lt' ↦ 'OLT'
         | 'le' ↦ 'OLE'
         };
  if(fp) return 'fcmp'(llop, l, r) else
       return 'icmp'(llop, l, r)
}
```

A helper function which returns a constant "1" of a given numeric type

```
function clike_c_one(tp)
 visit:clike2(lltype:tp) { deep lltype {
    integer ↦ 'val'('integer'(1,itype))
  | real ↦    'val'('float'("1.0",rtype))
  | ptr  ↦    'val'('integer'(1,'i32'))
  | else ↦    ccerror('CLIKE:PREPOSTERR'(tp))
 }}
```

A helper function which translates a prefix or a postfix operation

```
function clike_c_prepost(op, tp, varnm)
{
  bop =  case op { 'inc' | '++' ↦ 'Add'
                 | 'dec' | '--' ↦ 'Sub'};
  'storevar'(varnm, tp : 'binary'(bop, tp : 'loadvar'(varnm), tp : clike_c_one(tp)))
}
```

A helper function: fix a break target

```
function clike_fix_break(c, tgt)
  visit:clike3(llstmt2: c)
    { deep llstmt2 { break ↦ 'br_label'(tgt) | else ↦ node }}
```

An integer zero constant

```
define clike_zero = '_':'val'('integer'(0));
function clike_array_access(ar,idxs,node)
     match car(ar) with
          arg(t) ↦
             t: 'getelementptr'(ar,@idxs)
        | ptr(array(t,@_)) ↦
             'ptr'(t): 'getelementptr'(ar,clike_zero,@idxs)
        | ptr(ptr(t)) ↦
             'ptr'(t): 'getelementptr'('ptr'(t):'load'(ar),@idxs)
        | else ↦ ccerror('CLIKE:ARRAY-ACCESS-TYPE'(node))

function clike_elt_access(e,fldnm,node)
     match car(e) with
          arg(struct(@_)) ↦
             '_': 'getelementptr'(e,'_':
                       'val'('integer'(
                            clike_fieldnumber(car(e), fldnm))))
        | ptr(struct(@_)) ↦
             '_': 'getelementptr'(e,clike_zero,'_':
```

```
                'val'('integer'(
                        clike_fieldnumber(car(e), fldnm))))
    | else ↦ ccerror('CLIKE:FIELD-ACCESS-TYPE'(node))
```

5.1 A compilation pass: clike2→clike3

```
function clike_precompile(env, c)
  visit:clike2(llcode: c)
  { deep llcode {
       begin ↦ node
     | label ↦ 'begin'('br_label'(lbl),'label'(lbl))
     | vardef ↦ {
         match tp with
           array(eltt,@_) ↦ { // perform a decay immediately
            ntp = 'ptr'(eltt);
            name0 = gensym(); name1=gensym();
            'begin'(
              'set'(name0, clike_make_ref_type(tp) :
                           'alloca'(clike_ca_type(env, tp))),
              'set'(name, clike_make_ref_type(ntp) :
                           'alloca'(clike_c_type(env, ntp))),
              'store'('val'('var'(name)), clike_make_ref_type(ntp) :
                           'getelementptr'('_':
                               'val'('var'(name0)),
                                   clike_zero, clike_zero)))
           }
         | else ↦ {
            'set'(name, clike_make_ref_type(tp) :
                           'alloca'(clike_c_type(env, tp)))
           }
       }
     | set ↦ {
        match cdr(l) with
          val(vv) ↦ 'store'('val'(vv), e)
        | else ↦ {
          ptr = gensym();
          'begin'('set'(ptr, l),
                'store'('val'('var'(ptr)), e))}}
     | expr ↦ {
        match car(e) with
          ['void'] ↦ 'set'("",e)
        | else ↦ {
          dummy = gensym();
          'set'(dummy, e)}
       }
```

```
| return ↦ 'ret'(e)
| vreturn ↦ 'vret'()
| goto ↦ 'br_label'(lbl)
| do ↦ symbols(cnt, rep) {
  'begin'(
      'br_label'(rep),
   'label'(rep),
      clike_fix_break(body,cnt),
      'br'(cnd,rep,cnt),
   'label'(cnt))
  }
| while ↦ symbols(cnt, nxt, rep) {
  'begin'(
      'br_label'(rep),
   'label'(rep),
      'br'(cnd,nxt,cnt),
   'label'(nxt),
      clike_fix_break(body,cnt),
      'br_label'(rep),
   'label'(cnt))
  }
| for ↦ symbols(stepdummy, cnt, nxt, rep) {
   'begin'(
      @init,
      'br_label'(rep),
    'label'(rep),
      'br'(cnd, nxt, cnt),
    'label'(nxt),
      clike_fix_break(body,cnt),
      step,
      'br_label'(rep),
    'label'(cnt)
   )}
| if3 ↦ symbols(l1,l2,cnt) {
   'begin'(
       'br'(e,l1,l2),
      'label'(l1),
        tr,
        'br_label'(cnt),
      'label'(l2),
        fl,
        'br_label'(cnt),
      'label'(cnt)
    )}
| if2 ↦ symbols(l1,cnt) {
   'begin'(
```

```
          'br'(e,l1,cnt),
        'label'(l1),
          tr,
          'br_label'(cnt),
        'label'(cnt)
      )}
| switch ↦ symbols(l1,cnt,exit) {
    lbls = map o in opts do gensym();
    oz = collector(la, lg) {
           do lbloop(o = opts, l = lbls) {
      if(o) {la([car(o);car(l);if(cdr(l)) cadr(l) else exit]);
            lbloop(cdr(o),cdr(l))}
        };
    lg()
     };
    'begin'('switch'(e,cnt,map [[c;action];l;nxtl] in oz do
                [cadr(clike_c_const(env, c, car(e)));l]),
      @map append [[c;action];l;nxtl] in oz do {
       ['label'(l);
          clike_fix_break(action,exit);
          'br_label'(nxtl)
       ]},
      'label'(cnt),
       @dflt,
       'br_label'(exit),
      'label'(exit)
    )}
| passexpr ↦ ccerror('CLIKE:SHOULD-NOT-BE-HERE'(node))
| break ↦ 'break'()
| else ↦ ccerror('CLIKE:NOT-IMPLEMENTED-YET'(node))
};
deep lloexpr {
   call ↦ 'call'(id,@args)
| callptr ↦ 'callptr'(fn,@args)
| stdcallpfx ↦ {match cdr(e) with
                  'callptr'(fn,@args) ↦ 'callptrstd'(fn,@args)
                | else ↦ e}
| bin ↦ clike_binop_llvm(op,car(l), l, r)
| compop ↦ clike_compop_llvm(op, car(l), l, r)
| un ↦ clike_c_unop(env, op, e)
| tri ↦ symbols(tmp, l1, l2, next) {
    tp = car(tr);
    'liftstatements'('begin'('set'(tmp, clike_make_ref_type(tp) :
                                          'alloca'(clike_c_type(env, tp))),
                             'br'(cnd, l1, l2),
                           'label'(l1),
```

```
                        'storevar'(tmp, tr),
                        'br_label'(next),
                      'label'(l2),
                        'storevar'(tmp, fl),
                        'br_label'(next),
                      'label'(next)), tp : 'loadvar'(tmp))}
| typecast ↦ {cvop = clike_convop_llvm(t, car(e));
            if(cvop)
              'convop'(cvop, e,
                       clike_c_type(env, t))
            else cdr(e) }
| pre ↦ symbols(tmp) {
   tp = if(vtyp) clike_deref_type(car(vtyp)) else '_';
   match cdr(v) with
     val(var(vv)) ↦
       'liftstatements'('begin'(
                        clike_c_prepost(op, tp, vv)),
                      tp : 'loadvar'(vv))
    | else ↦
       'liftstatements'('begin'('set'(tmp, v),
                        clike_c_prepost(op, tp, tmp)),
                      tp : 'loadvar'(tmp))}
| post ↦ symbols(tmp, tstor) {
   tp = if(vtyp) clike_deref_type(car(vtyp)) else '_';
   match cdr(v) with
     val(var(vv)) ↦
       'liftstatements'('begin'(
                        'set'(tstor,tp : 'loadvar'(vv)),
                        clike_c_prepost(op, tp, vv)),
                    tp : 'val'('var'(tstor)))
    | else ↦
       'liftstatements'('begin'('set'(tmp,v),
                        'set'(tstor,tp : 'loadvar'(tmp)),
                        clike_c_prepost(op, tp, tmp)),
                    tp : 'val'('var'(tstor)))}
| inblock ↦ 'liftstatements'(c, r)
| eset ↦ symbols(tmp, tstor) {
   tp = car(v);
   match cdr(v) with
     val(vv) ↦
       'liftstatements'('begin'(
                        'set'(tstor,e),
                        'store'('val'(vv), tp : 'val'('var'(tstor)))),
                    tp : 'val'('var'(tstor)))
   | else ↦
       'liftstatements'('begin'('set'(tmp,v),
```

```
                        'set'(tstor,e),
                        'storevar'(tmp, tp : 'val'('var'(tstor)))),
                    tp : 'val'('var'(tstor)))}
| modop ↦ symbols(tmp, tstor) {
    tp = car(l);
    'liftstatements'('begin'('set'(tstor, l),
                        'set'(tmp, tp :
                            clike_binop_llvm(op, tp, tp:'load'(l),
                                                r)),
                        'storevar'(tstor,tp:'val'('var'(tmp)))),
                    tp : 'val'('var'(tmp)))
  }
| logand ↦ symbols(reslt, cnt) {
    'liftstatements'('begin'('set'(reslt, '_':
                            'alloca'('integer'('i32'))),
                        @map append es do symbols(tv, nxt1) {[
                            'set'(tv, es);
                            'storevar'(reslt, '_':'val'('var'(tv)));
                            'br'('_':'val'('var'(tv)), nxt1, cnt);
                            'label'(nxt1)
                          ]},
                        'br_label'(cnt),
                        'label'(cnt)
                        ),'_':'loadvar'(reslt))}
| logor ↦ symbols(reslt, cnt) {
    'liftstatements'('begin'('set'(reslt, '_':
                            'alloca'('integer'('i32'))),
                        @map append es do symbols(tv, nxt1) {[
                            'set'(tv, es);
                            'storevar'(reslt, '_':'val'('var'(tv)));
                            'br'('_':'val'('var'(tv)), cnt, nxt1);
                            'label'(nxt1)
                          ]},
                        'br_label'(cnt),
                        'label'(cnt)
                        ),'_':'loadvar'(reslt))}
| const ↦ clike_c_const(env, c, caar(stack))
| globstring ↦ 'stringtmp'(s)
| var ↦ 'loadvar'(nm)
| arg ↦ 'val'('var'(nm))
| glob ↦ 'load'('_':'val'('global'(nm)))
| globfun ↦ 'load'('_':'val'('globalfun'(nm)))
| array ↦ 'load'(clike_array_access(ar,idxs,node))
| ref ↦ cdr(e)
| deref ↦ 'load'(e)
| getelt ↦ 'load'(clike_elt_access(e,fldnm,node))
```

```
    | sizeof ↦ 'val'('sizeof'(clike_c_type(env, t)))
    };
    // lvalues are different, they are compiled into a code which gives a
    //  pointer instead of a value
    deep olvalue {
      var ↦ 'val'('var'(nm)) // all local vars are pointers in llvm
    | arg ↦ 'val'('var'(nm))
    | glob ↦ 'val'('global'(nm))
    | globfun ↦ 'val'('globalfun'(nm))
    | array ↦ cdr(clike_array_access(ar,idxs,node))
    | deref ↦ cdr(e)
    | getelt ↦ cdr(clike_elt_access(e,fldnm,node))
    };
  }
```

A helper compilation pass: gets rid of unnecessary (and impossible) redefinitions. In addition, 'loadvar' sugar is expanded.

```
function clike_fix_sets(c)
{  ren = mkhash();
   visit:clike3(llstmt2: c)
   {
     deep llstmt2 {
       set ↦ {
         match cdr(e) with
           'val'(v) ↦ {
             ren /! nm <- v;
             'nop'()
           }
         | else ↦ node
       }
     | storevar ↦ {
         chk = ren/@ptr;
         vv = if(chk) chk else 'var'(ptr);
         return 'store'('val'(vv),e)
       }
     | else ↦ node
     };
     deep llexpr1 {
     | loadvar ↦ {
         chk = ren /@ id;
         v = if(chk) chk else 'var'(id);
         return 'load'('_':'val'(v))
       }
     | else ↦ node
```

```
    };
    deep llval {
      var ↦ {
        chk = ren /@ nm;
        if(chk) chk else node
      }
    | else ↦ node
    };
  }
}
```

A helper function for a next compilation pass: flattens an expression tree, leaving topmost expression intact

```
function clike_lift_2(add, expr)
  visit:clike3(llexpr1_deref: expr) {
    llexpr1 as llexpr1_deref {
      | liftstatements ↦ cdr(e)
      | else ↦ node
    };
    once llstmt2 : ∀ add(clike_lift_1(node));
    deep llexpr1 {
      val ↦ node
    | liftstatements ↦ cdr(e)
    | else ↦ {
        newnm = gensym();
        add('set'(newnm,'_':clike_lift_2(add, node)));
        return 'val'('var'(newnm))
      }
    }
  }
```

The same as above, but the topmost expression is also lifted as a variable binding

```
function clike_lift_3(add, expr)
  visit:clike3(llexpr1: expr) {
    once llstmt2 : ∀ add(clike_lift_1(node));
    deep llexpr1 {
      val ↦ node
    | liftstatements ↦ cdr(e)
    | else ↦ {
        newnm = gensym();
        add('set'(newnm,'_':clike_lift_2(add, node)));
        return 'val'('var'(newnm))
      }}}
```

A compilation pass: flatten all the expression trees

```
function clike_lift_1(code)
  collector(add, get) {
    do loop(c = code)
      iter:clike3(llstmt2: c) { once llstmt2 {
          begin ↦ iter es do loop(es)
        | set ↦ {
          add(visit:clike3(llstmt2: node) {
                once llexpr1 : ∀ clike_lift_2(add, node)
              })}
        | else ↦ {
          add(visit:clike3(llstmt2: node) {
                once llexpr1 : ∀ clike_lift_3(add, node)
              })
        }
      }};
    return 'begin'(@get())
  }
```

A helper compilation pass: remove 'val' nodes, remove the remaining types annotations, flatten nested 'begin' sequences, fail on any remaining breaks

```
function clike_cleanup(c)
 collector(aladd, alget) {
  rest =
   visit:clike3(llstmt2: c)
    { deep llexpr2 : e;
      deep llexpr1 { val ↦ v
                   | liftstatements ↦ ccerror('CLIKE:BROKEN-PASS'(node))
                   | else ↦ node };
      deep llstmt2 {
          begin ↦ map append es do es
        | break ↦ ccerror('CLIKE:BREAK-OUT-OF-CONTEXT'())
        | nop ↦ ∅
        | set ↦ {
            match e with
              'alloca':_ ↦ {aladd(node);∅}
            | stringtmp(s) ↦ ['setstring'(nm,s)]
            | else ↦ [node]
          }
        | else ↦ [node]
      };
    };
```

```
  return alget()⊕rest;
  }

function clike_rettype_voidp(t)
  match t with
     void() ↦ true
   | else ↦ ∅
```

A final compilation pass: separate the basic blocks. After this pass, the code is ready to be translated into LLVM IR.

```
function clike_basicblocks(rettype, code)
  collector(badd, bget) {
    voidp = clike_rettype_voidp(rettype);
    nextbblock(name) = collector(add, get) { add:λ() {name:get()} };
    do loop(c = code, bb = nextbblock("entry"), empt = true) {
      if(%null?(c)) {
        if(not(%null?(bb))) { // if we're here, a return must be injected
          if(not(voidp)) {
              (car(bb))('ret'('zero'(rettype)));
              badd((cdr(bb))());
          } else { (car(bb))('vret'());
                   badd((cdr(bb))()); }
        }
      }
      else case caar(c) {
        'br' | 'br_label' | 'switch' | 'indirectbr' | 'ret' | 'vret' ↦
          {
            if(bb) {
              (car(bb))(car(c));
              badd((cdr(bb))());
            };
            loop(cdr(c),∅,true)
          }
       | 'label' ↦ {
            loop(cdr(c), nextbblock(%S<<(cadr(car(c)))), true)
          }
       | else ↦ {
            if(%null?(bb)) { // Unreacheable code
           loop(c, nextbblock(%S<<(gensym())), true)
          } else {
              (car(bb))(car(c));
              loop(cdr(c),bb,∅)
          }
```

```
      }
     }
   };
   return bget();
 }
```

6 A toplevel compiler

A compilation frontend for function bodies: binds all the passes together Pipeline is following: types propagation → tree compilation → values redefs elimination → tree flattening → values redefs elimination → metadata elimination → basic blocks extraction

```
function clike_compile_code(toploop, env, code, rettype)
{
  clike_dbg(1, "S0:",code);

  step1 = clike_types(env, code, toploop);   clike_dbg(1,"S1:",step1);
  step2 = clike_precompile(env, step1);      clike_dbg(2,"S2:",step2);
  step3 = clike_fix_sets(step2);             clike_dbg(3,"S3:",step3);
  step3_1 = clike_lift_1(step3);             clike_dbg(4,"S3_1:",step3_1);
  step4 = clike_fix_sets(step3_1);           clike_dbg(5,"S4:",step4);
  step5 = clike_cleanup(step4);
  step6 = clike_basicblocks(rettype,step5); clike_dbg(6,"S6:",step6);

  return step6;
}
```

A compilation frontend for toplevel definitions. It is possible that a new toplevel expression is lifted, so an external top loop function should be provided.

```
function clike_compile(etoploop, topenv, top)
 collector(topsadd, topsget)
 {
  toploop(env, t) = { iter i in etoploop(t) do topsadd(i) };
  rcode =
   visit:clike(lltoplev: top)
   {
     once topident : clike_env_name_mangle(topenv:∅, node);
     once llvarname { v ↦ mk:node(name=clike_env_name_mangle(topenv:∅, name))
                    | else ↦ node };
     once llcode : ∀ node; // stop here, do not touch llvarnames inside
     deep lltoplev {
        begin ↦ map append es do es
      | typedef ↦ {  clike_env_defalias(topenv, name, tp);
                     clike_dbg(0,"Top:",#'(typedef: ,tp ,name));
```

```
                ['comment'('clike'(node))]}
| xfunc ↦ {clike_env_deffunction(topenv, name, va, ret, args); ∅}
| xglobal ↦ {clike_env_defglobal(topenv, name, tp); ∅}
| efunc ↦ {clike_env_deffunction(topenv, name, va, ret, args);
          clike_dbg(0,"Top:",#'(efunc: ,name ,ret ,@args ,va));
          env = topenv:mkhash();
          cc1 = if(cc) ['stdcall'] else ∅;
          ['comment'('clike'('xfunc'(ret,name, va,@args)));
           'function'(cc1,name, clike_c_type(env, clike_env_unitype(env, ret)), va,
                      map [tp;'v'(nm)] in args do {
                        [clike_c_type(env,clike_env_unitype(env, tp)); nm]
                      })]
         }
| global ↦ {clike_env_defglobal(topenv, cadr(name), tp);
            clike_dbg(0,"Top:",#'(global: ,tp ,name));
            env = topenv:mkhash();
            gtp = clike_c_type(env,clike_env_unitype(env, tp));
            return ['global'( %S<<(cadr(name)),
                      gtp,
                      'zero'(gtp)
                    )]}
| eglobal ↦ {clike_env_defglobal(topenv, cadr(name), tp);
             clike_dbg(0,"Top:",#'(global: ,tp ,name));
             env = topenv:mkhash();
             return ['comment'('clike'('xglobal'(tp, cadr(name))));
                     'eglobal'( %S<<(cadr(name)),
                                clike_c_type(env,clike_env_unitype(env, tp)) )]
            }
| cfunc ↦ {
          env = clike_local_env(topenv, args);
          clike_dbg(0,"Top:",#'(cfunc: ,name ,ret ,@args));
          clike_env_deffunction(topenv, name, va, ret, args);
          rett =  clike_env_unitype(env, ret);
          cbody = clike_compile_code(toploop,env,body,
                                     clike_c_type(env,rett));
          clike_env_savebody(topenv, name, body, cbody);
          cc1 = if(cc) ['stdcall'] else ∅;
          ['comment'('clike'('xfunc'(ret,name,va,@args)));
           'function'(cc1, name, clike_c_type(env,rett), va,
                      map [tp;'v'(nm)] in args do {
                        [clike_c_type(env,clike_env_unitype(env, tp)); nm]
                      },
                      @cbody)]
  }
| else ↦ ccerror('CLIKE:NOT-IMPLEMENTED-YET'(node))
}
```

```
    };
  iter rcode do topsadd(rcode);
  return topsget();
}

function clike_to_llvm_inner(env, cltops)
{
  cl1 = map t in cltops do clike_expand_macros_top(env, clike_expand_core(t));
  cl2 = map append t in cl1 do clike_compile(λ(t) {clike_to_llvm_inner(env, [t])}, env, t);
  return cl2;
}


function clike_to_llvm(topenv, tops)
{
  try {
  try clike_to_llvm_inner(topenv, tops)
  catch (t_MBaseException e) {
     println("Compiler error:");
     println(mbaseerror(e));
     println(%->s(e));
     return ∅
  }} catch (t_Exception e) {
     println(%->s(e));
     return ∅
  }
}
```