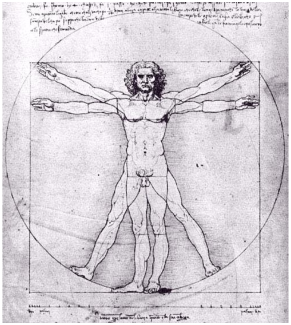
The MicroC Compiler

Stephen A. Edwards

Columbia University

Spring 2021



The MicroC Language

A very stripped-down dialect of C

Functions, global variables, and most expressions and statements, but only integer, float, and boolean values.

```
/* The GCD algorithm in MicroC */

int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

int main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
  return 0;
}
```

Scanning and Parsing

Tokenize and parse to produce
an Abstract Syntax Tree

The first part of any compiler or interpreter

The Scanner (scanner.mll)

```
{ open Microcparse }
let digit = ['0' - '9']

rule token = parse
  [' ' '\t' '\r' '\n'] { token   lexbuf }
| "/*"                 { comment lexbuf }
| "if"     { IF }       | '(' { LPAREN } | '='  { ASSIGN }
| "else"   { ELSE }     | ')' { RPAREN } | "==" { EQ }     | ">"  { GT }
| "for"    { FOR }      | '{' { LBRACE } | "!=" { NEQ }    | ">=" { GEQ }
| "while"  { WHILE }    | '}' { RBRACE } | '<'  { LT }     | "&&" { AND }
| "return" { RETURN }   | ';' { SEMI }   | "<=" { LEQ }    | "||" { OR }
| "int"    { INT }      | '+' { PLUS }   | ','  { COMMA }  | "!"  { NOT }
| "bool"   { BOOL }     | '-' { MINUS }  | "true"   { BLIT(true)  }
| "float"  { FLOAT }    | '*' { TIMES }  | "false"  { BLIT(false) }
| "void"   { VOID }     | '/' { DIVIDE }
| digit+ as lxm { LITERAL(int_of_string lxm) }
| digit+ '.' digit* (['e' 'E'] ['+' '-']? digits)? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*  as lxm { ID(lxm)   }
| eof { EOF }
| _ as ch { raise (Failure("illegal character " ^ Char.escaped ch)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

The AST (ast.ml)

```
type op   = Add | Sub | Mult | Div | Equal | Neq | Less | Leq
          | Greater | Geq | And | Or
type uop  = Neg | Not
type typ  = Int | Bool | Float | Void
type bind = typ * string

type expr = Literal of int | Fliteral of string | BoolLit of bool
          | Id      of string
          | Binop   of expr * op * expr | Unop of uop * expr
          | Assign  of string * expr
          | Call    of string * expr list
          | Noexpr
type stmt = Block  of stmt list
          | Expr   of expr
          | Return of expr
          | If     of expr * stmt * stmt
          | For    of expr * expr * expr * stmt
          | While  of expr * stmt
type func_decl = { typ      : typ;
                   fname    : string;
                   formals  : bind list;
                   locals   : bind list;
                   body     : stmt list;  }

type program = bind list * func_decl list
```

The Parser (microcparse.mly)

```
%{ open Ast %}
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT EQ
%token NEQ LT LEQ GT GEQ AND OR RETURN IF ELSE
%token FOR WHILE INT BOOL FLOAT VOID
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT

%%
```

Declarations

```
program: decls EOF { $1 }

decls: /* nothing */ { ([], [])               }
     | decls vdecl   { (($2 :: fst $1), snd $1) }
     | decls fdecl   { (fst $1, ($2 :: snd $1)) }

fdecl: typ ID LPAREN formals_opt RPAREN
       LBRACE vdecl_list stmt_list RBRACE {
     { typ = $1; fname = $2; formals = List.rev $4;
       locals = List.rev $7; body = List.rev $8 } }

formals_opt:  /* nothing */ { [] }
           | formal_list    { $1 }

formal_list: typ ID                       { [($1,$2)]     }
           | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ: INT   { Int   }    | BOOL  { Bool  }
   | FLOAT { Float }    | VOID  { Void  }

vdecl_list: /* nothing */    { [] }
          | vdecl_list vdecl { $2 :: $1 }

vdecl: typ ID SEMI { ($1, $2) }
```

Statements

```
stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI                            { Expr $1               }

  | RETURN expr_opt SEMI                 { Return $2             }

  | LBRACE stmt_list RBRACE              { Block(List.rev $2)    }

  | IF LPAREN expr RPAREN stmt %prec NOELSE
                                         { If($3, $5, Block([])) }

  | IF LPAREN expr RPAREN stmt ELSE stmt
                                         { If($3, $5, $7)        }

  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                         { For($3, $5, $7, $9)   }

  | WHILE LPAREN expr RPAREN stmt        { While($3, $5)         }
```

Expressions

```
expr:
    LITERAL                    { Literal($1)              }
  | FLIT                       { Fliteral($1)             }
  | BLIT                       { BoolLit($1)              }
  | ID                         { Id($1)                   }
  | expr PLUS   expr           { Binop($1, Add,   $3)     }
  | expr MINUS  expr           { Binop($1, Sub,   $3)     }
  | expr TIMES  expr           { Binop($1, Mult,  $3)     }
  | expr DIVIDE expr           { Binop($1, Div,   $3)     }
  | expr EQ     expr           { Binop($1, Equal, $3)     }
  | expr NEQ    expr           { Binop($1, Neq,   $3)     }
  | expr LT     expr           { Binop($1, Less,  $3)     }
  | expr LEQ    expr           { Binop($1, Leq,   $3)     }
  | expr GT     expr           { Binop($1, Greater, $3)   }
  | expr GEQ    expr           { Binop($1, Geq,   $3)     }
  | expr AND    expr           { Binop($1, And,   $3)     }
  | expr OR     expr           { Binop($1, Or,    $3)     }
  | MINUS expr %prec NOT       { Unop(Neg, $2)            }
  | NOT expr                   { Unop(Not, $2)            }
  | ID ASSIGN expr             { Assign($1, $3)           }
  | ID LPAREN args_opt RPAREN
                               { Call($1, $3)             }
  | LPAREN expr RPAREN         { $2                       }
```

Expressions concluded

```
expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

args_list:
    expr                    { [$1] }
  | args_list COMMA expr { $3 :: $1 }
```

Testing with menhir

```
$ menhir --interpret --interpret-show-cst microcparse.mly
INT ID LPAREN RPAREN LBRACE ID LPAREN LITERAL RPAREN SEMI RBRACE EOF
ACCEPT
```

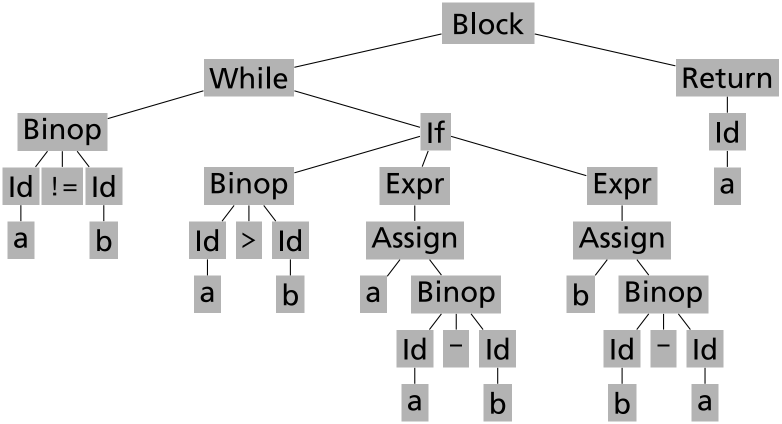
```
[program:
  [decls:
    [decls:]
    [fdecl:
      [typ: INT]
      ID
      LPAREN
      [formals_opt:]
      RPAREN
      LBRACE
      [vdecl_list:]
      [stmt_list:
        [stmt_list:]
        [stmt:
          [expr:
            ID
            LPAREN
            [actuals_opt: [actuals_list: [expr: LITERAL]]]
            RPAREN
          ]
          SEMI
        ]
      ]
      RBRACE
    ]
  ]
  EOF
]
```

```
int main() {
  print(42);
}
```

AST for the GCD Example

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}
```

```
typ = Int
fname = gcd
formals = [Int a; Int b]
locals = []
body =
```



AST for the GCD Example

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}
```

```
typ = Int
fname = gcd
formals = [Int a; Int b]
locals = []
body =
```

```
[While (Binop (Id a) Neq (Id b))
       (Block [(If (Binop (Id a) Greater (Id b))
                   (Expr (Assign a
                           (Binop (Id a) Sub (Id b))))
                   (Expr (Assign b
                           (Binop (Id b) Sub (Id a)))))
              ]),
 Return (Id a)]
```

Testing the Parser: AST Pretty Printing

ast.ml has pretty-printing functions; invoke with -a

```
$ ocamlbuild microc.native
Finished, 16 targets (0 cached) in 00:00:00.
$ ./microc.native -a tests/test-gcd.mc
int main()
{
print(gcd(2, 14));
print(gcd(3, 15));
print(gcd(99, 121));
return 0;
}

int gcd(a,b)
{
while (a != b) {
if (a > b)
a = a - b;
else
b = b - a;
}
return a;
}
```

Static Semantic Checking

Walk over the AST
Verify each node
Establish existence of each identifier
Establish type of each expression
Validate statements in functions

Static Semantic Analysis

Lexical analysis: Each token is valid?

```
if i 3 "This"                 /* valid Java tokens */
#a1123                        /* not a token */
```

Syntactic analysis: Tokens appear in the correct order?

```
for ( i = 1 ; i < 5 ; i++ ) 3 + "foo";  /* valid Java syntax */
for break                               /* invalid syntax */
```

Semantic analysis: Names used correctly? Types consistent?

```
int v = 42 + 13;    /* valid in Java (if v is new) */
return f + f(3);    /* invalid */
```

What To Check

Examples from Java:

Verify names are defined and are of the right type.

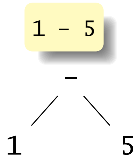
```
int i = 5;
int a = z;    /* Error: cannot find symbol */
int b = i[3]; /* Error: array required, but int found */
```

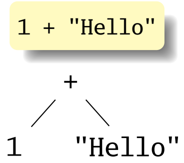
Verify the type of each expression is consistent.

```
int j = i + 53;
int k = 3 + "hello";   /* Error: incompatible types */
int l = k(42);         /* Error: k is not a method */
if ("Hello") return 5; /* Error: incompatible types */
String s = "Hello";
int m = s;             /* Error: incompatible types */
```

How To Check Expressions: Depth-first AST Walk

check: environment → node → typedNode





check(–)
- check(1) = 1 : int
- check(5) = 5 : int
- int – int = int
- = 1 – 5 : int

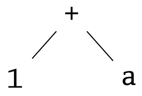
check(+)
- check(1) = 1 : int
- check(“Hello”) = “Hello” : string
- FAIL: Can’t add int and string

Ask yourself: at each kind of node, what must be true about the nodes below it? What is the type of the node?

How To Check Symbols

check: environment → node → typedNode

```
1 + a
```



check(+)
 check(1) = 1 : int
 check(a) = a : lookup(a) = a : int
 int + int = int
 = 1 + a : int

The key operation: determining the type of a symbol.

The environment provides a “symbol table” that holds information about each in-scope symbol.

Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```

Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```

$\boxed{x \mapsto \textbf{int}}$

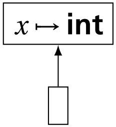
Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a "block": New symbol table; point to previous

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```



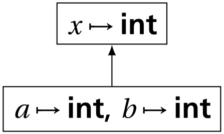
Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```



$$\begin{tikzcd}
\boxed{x \mapsto \textbf{int}} \\
\boxed{a \mapsto \textbf{int},\ b \mapsto \textbf{int}} \arrow[u]
\end{tikzcd}$$

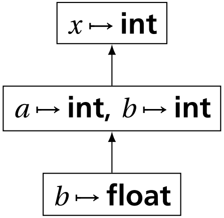
Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```



$$\begin{tikzcd} \boxed{x \mapsto \textbf{int}} \\ \boxed{a \mapsto \textbf{int},\ b \mapsto \textbf{int}} \arrow[u] \\ \boxed{b \mapsto \textbf{float}} \arrow[u] \end{tikzcd}$$

Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous
- Reach an identifier: lookup in chain of tables

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```

$$\begin{tikzcd}
\boxed{x \mapsto \textbf{int}} \\
\boxed{a \mapsto \textbf{int},\ b \mapsto \textbf{int}} \arrow[u] \\
\boxed{b \mapsto \textbf{float}} \arrow[u] \\
\boxed{i \mapsto \textbf{int},\ b \mapsto \textbf{int}} \arrow[u]
\end{tikzcd}$$

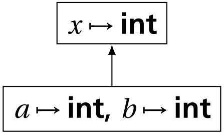
Symbol Tables by Example: C-style

Scope: area of program where a name has meaning

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous
- Reach an identifier: lookup in chain of tables
- Leave a block: Local symbol table disappears

```
int x;
int main()
{
  int a = 1;
  int b = 1;
  {
    float b = 2;
    for (int i = 0; i < b; i++)
    {
      int b = i;
    }
  }
  b + x;
}
```


$$\begin{tikzcd} \boxed{x \mapsto \textbf{int}} \\ \boxed{a \mapsto \textbf{int},\ b \mapsto \textbf{int}} \arrow[u] \end{tikzcd}$$

The Type of Types

Need an OCaml type to represent the type of something in your language.

For MicroC, it’s simple (from ast.ml):

```
type typ  = Int | Bool | Float | Void
```

For a language with integer, structures, arrays, and exceptions:

```
type ty = (* can’t call it "type" since that’s reserved *)
     Void
   | Int
   | Array of ty * int                            (* type, size *)
   | Exception of string
   | Struct of string * ((string * ty) array) (* name, fields *)
```

Implementing a Symbol Table and Lookup

It’s a structured dictionary. A map, hash, or some combination is typical.

lookup: string → type.

```
module StringMap = Map.Make(String)

type symbol_table = {
  (* Variables bound in current block *)
  variables : ty StringMap.t
  (* Enclosing scope *)
  parent : symbol_table option;
}
```

```
let rec find_variable (scope : symbol_table) name =
  try
     (* Try to find binding in nearest block *)
     StringMap.find name scope.variables
  with Not_found -> (* Try looking in outer blocks *)
    match scope.parent with
      Some(parent) -> find_variable parent name
    | _ -> raise Not_found
```

Translation Environments

Whether an expression/statement/function is correct depends on its context. Represent this as an object with named fields since you will invariably have to extend it.

An environment type for a C-like language:

```
type translation_environment = {
    scope : symbol_table;     (* symbol table for vars *)

    return_type : ty option; (* Function's return type *)
    in_switch : bool;         (* if we are in a switch stmt *)
    labels : string list ;    (* labels on statements *)
}
```

A Static Semantic Checking Function

check: ast → sast

Converts a raw AST to a “semantically checked AST”

Names and types resolved

AST:

```
type expr =
    Literal of int
  | Id of string
  | Call of string * expr list
  | ...
```

⇓

SAST:

```
type expr_detail =
    SLiteral of int
  | SId of string
  | SCall of string * sexpr list
  | ...

type sexpr = expr_detail * typ
```

The MicroC Semantic Checker

The Semantically-Checked AST

```
open Ast
type sexpr = typ * sx    (* The one important change *)
and sx = SLiteral  of int
       | SFliteral of string
       | SBoolLit  of bool
       | SId       of string
       | SBinop    of sexpr * op * sexpr
       | SUnop     of uop * sexpr
       | SAssign   of string * sexpr
       | SCall     of string * sexpr list
       | SNoexpr
type sstmt = SBlock  of sstmt list
           | SExpr   of sexpr
           | SReturn of sexpr
           | SIf     of sexpr * sstmt * sstmt
           | SFor    of sexpr * sexpr * sexpr * sstmt
           | SWhile  of sexpr * sstmt
type sfunc_decl = { styp     : typ;
                    sfname   : string;
                    sformals : bind list;
                    slocals  : bind list;
                    sbody    : sstmt list; }
type sprogram = bind list * sfunc_decl list
```

The MicroC Semantic Checker (semant.ml)

```
open Ast
open Sast
module StringMap = Map.Make(String)

(* Some type definitions to clarify signatures *)
type func_symbol = func_decl StringMap.t

(* Semantic checking of the AST. Returns a semantically
   checked program (globals, SAST) if successful;
   throws an exception if something is wrong. *)

let check (globals, functions) =

  (* ... many lines of code .. *)

  in (globals, List.map check_function functions)
```

The check_binds helper function

Verify a list of bindings has no “void” type or duplicate names.

Used for globals, formal parameters, and local variables.

```
let check_binds (kind : string) (binds : bind list) =
  List.iter (function
      (Void, b) -> raise
                     (Failure ("illegal void " ^ kind ^ " " ^ b))
    | _ -> ()) binds;
  let rec dups = function
      [] -> ()
    |   ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
        raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
    | _ :: t -> dups t
  in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
in
```

Global Variables, Built-in Functions

```
(**** Check global variables ****)

check_binds "global" globals;

(**** Check functions ****)

(* Collect function declarations for built-in functions: no bodies *)
let built_in_decls =
  let add_bind map (name, ty) = StringMap.add name {
    typ = Void;
    fname = name;
    formals = [(ty, "x")];
    locals = []; body = [] } map
  in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                               ("printb", Bool);
                                               ("printf", Float);
                                               ("printbig", Int) ]
in
```

MicroC has 4 built-in functions, *print*, *printb*, *printf*, and *printbig*; this is an easy way to check them. Your compiler should have very few exceptions like this.

Function Symbol Table and “main”

```
(* Add function name to symbol table *)
let add_func map fd =
  let built_in_err = "function " ^ fd.fname ^ " may not be defined"
  and dup_err = "duplicate function " ^ fd.fname
  and make_err er = raise (Failure er)
  and n = fd.fname (* Name of the function *)
  (* Prohibit duplicate names or redefinitions of built-ins *)
  in match fd with
       _ when StringMap.mem n built_in_decls -> make_err built_in_err
     | _ when StringMap.mem n map -> make_err dup_err
     | _ ->  StringMap.add n fd map
in

(* Collect all function names into one symbol table *)
let function_decls = List.fold_left add_func built_in_decls functions
in

(* Return a function from our symbol table *)
let find_func s =
  try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

(* Ensure "main" is defined *)
let _ = find_func "main" in
```

Check a Function

```
let check_function func =
  (* Make sure no formals or locals are void or duplicates *)
  check_binds "formal" func.formals;
  check_binds "local" func.locals;
```

A critical helper function for all kinds of assignments:

In the assignment *lvalue* = *rvalue*,
can the type of *rvalue* be assigned to *lvalue*?

In the call *f(…, arg*$_i$*, …)* where *f* has formals [..., *formal*$_i$, ...],
can *arg*$_i$ be assigned to *formal*$_i$?

```
  (* Raise an exception if the given rvalue type cannot
     be assigned to the given lvalue type *)
let check_assign lvaluet rvaluet err =
   if lvaluet = rvaluet then lvaluet else raise (Failure err)
in
```

Variable Symbol Table

What can happen when you refer to a variable?

What are MicroC's *scoping rules*?

```
int a;     /* Global variable */
int c;

void foo(int a) { /* Formal arg. */
  int b;  /* Local variable   */
  ... a = ... a ...
  ... b = ... b ...
  ... c = ... c ...
  ... d = ... d ...
}
```

```
(* Variable symbol table: type of each global, formal, local *)
let symbols = List.fold_left
               (fun m (t, n) -> StringMap.add n t m)
               StringMap.empty
               ( globals @ func.formals @ func.locals )
in

(* The key symbol table lookup operation *)
let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found ->
    raise (Failure ("undeclared identifier " ^ s))
in
```

Expressions

The expr function: return an SAST sexpr w/type

```
(* Return a semantically-checked expression, i.e., with a type *)
let rec expr = function
    Literal  l -> (Int,   SLiteral l)
  | Fliteral l -> (Float, SFliteral l)
  | BoolLit l  -> (Bool,  SBoolLit l)
  | Noexpr     -> (Void,  SNoexpr)
```

An identifier: does it exist? What is its type?

```
  | Id s       -> (type_of_identifier s, SId s)
```

Assignment: need to know the types of the *lvalue* and *rvalue*, and whether one can be assigned to the other.

```
  | Assign(var, e) as ex ->
      let lt = type_of_identifier var
      and (rt, e') = expr e in
      let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
        string_of_typ rt ^ " in " ^ string_of_expr ex
      in (check_assign lt rt err, SAssign(var, (rt, e')))
```

Unary Operators

What type is the argument?

```
| Unop(op, e) as ex ->
    let (t, e') = expr e in
    let ty = match op with
      Neg when t = Int || t = Float -> t
    | Not when t = Bool             -> Bool
    | _ -> raise (Failure ("illegal unary operator " ^
                           string_of_uop op ^ string_of_typ t ^
                           " in " ^ string_of_expr ex))
    in (ty, SUnop(op, (t, e')))
```

Binary Operators

Check the types of both operands:

```
| Binop(e1, op, e2) as e ->
    let (t1, e1') = expr e1
    and (t2, e2') = expr e2 in
    (* All binary operators require operands of the same type *)
    let same = t1 = t2 in
    (* Type depends on the operator and types of operands *)
    let ty = match op with
      Add | Sub | Mult | Div when same && t1 = Int   -> Int
    | Add | Sub | Mult | Div when same && t1 = Float -> Float
    | Equal | Neq            when same               -> Bool
    | Less | Leq | Greater | Geq
                 when same && (t1 = Int || t1 = Float) -> Bool
    | And | Or when same && t1 = Bool                -> Bool
    | _ -> raise (
        Failure ("illegal binary operator " ^
                 string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                 string_of_typ t2 ^ " in " ^ string_of_expr e))
    in (ty, SBinop((t1, e1'), op, (t2, e2')))
```

Function Calls

Number and type of formals and actuals must match

```
void foo(t1 f1, t2 f2) { ... }
```

```
... = ... foo(expr1, expr2) ...
```

The callsite behaves like

```
f1 = expr1;
f2 = expr2;
```

```
| Call(fname, args) as call ->
    let fd = find_func fname in
    let param_length = List.length fd.formals in
    if List.length args != param_length then
      raise (Failure ("expecting " ^ string_of_int param_length ^
                      " arguments in " ^ string_of_expr call))
    else let check_call (ft, _) e =
      let (et, e') = expr e in
      let err = "illegal argument found " ^ string_of_typ et ^
        " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
      in (check_assign ft et err, e')
    in
    let args' = List.map2 check_call fd.formals args
    in (fd.typ, SCall(fname, args'))
```

Statements

Make sure an expression is Boolean: used in *if*, *for*, *while*.

```
let check_bool_expr e =
  let (t', e') = expr e
  and err = "expected Boolean expression in " ^ string_of_expr e
  in if t' != Bool then raise (Failure err) else (t', e')
in
```

Checking a statement: make sure it is well-formed and return a semantically-checked statement (i.e., SAST.sstmt)

```
let rec check_stmt = function
    Expr e -> SExpr (expr e)
  | If(p, b1, b2) ->
      SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
  | For(e1, e2, e3, st) ->
      SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
  | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
```

Statements: Return

The type of the argument to *return* must match the type of the function.

```
      | Return e -> let (t, e') = expr e in
        if t = func.typ then SReturn (t, e')
        else raise (
          Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                   string_of_typ func.typ ^ " in " ^ string_of_expr e)
```

Statements: Blocks

Checking a block of statements is almost `List.iter stmt sl`, but LLVM does not like code after a return:

```
int main() {
   return 1;
   print(42); /* Illegal: code after a return */
}
```

```
      | Block sl ->
          let rec check_stmt_list = function
              [Return _ as s] -> [check_stmt s]
            | Return _ :: _   -> raise (Failure "nothing may follow a
            | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten
            | s :: ss         -> check_stmt s :: check_stmt_list ss
            | []              -> []
          in SBlock(check_stmt_list sl)
```

semant.ml: The Big Picture

```
let check (globals, functions) =

  (* check_binds, check globals,
     build and check function symbol table, check for main *)

  let check_function func =

    (* check formal and local bindings *)

    let rec expr = (* ... *) in

    let rec check_stmt = (* ... *)

  in { styp     = func.typ;
       sfname   = func.fname;
       sformals = func.formals;
       slocals  = func.locals;
       sbody    = match check_stmt (Block func.body) with
         SBlock(sl) -> sl
       | _ -> raise (Failure
             ("internal error: block didn't become a block?"))
  }

in (globals, List.map check_function functions)
```

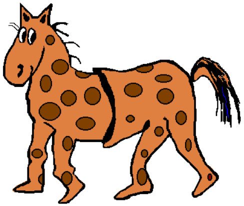
Code Generation

A Long K's Journey into Byte[†]

Compiler front end	Source code	
	↓	Parser/Semantic Analysis
	AST	
Compiler back end	↓	Intermediate code generation
	IR	
	↓	Optimization
	Assembly Code	
Assembler	↓	Assemble
	Relocatable Object Code	
Linker	↓	Link
	Executable	
Loader	↓	Relocate
	In-memory image	

†Apologies to O'Neill

Compiler Frontends and Backends



The front end focuses on *analysis*:

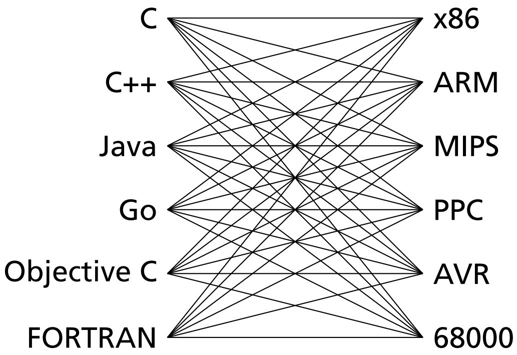
- Lexical analysis
- Parsing
- Static semantic checking
- AST generation

The back end focuses on *synthesis*:

- Translation of the AST into intermediate code
- Optimization
- Generation of assembly code

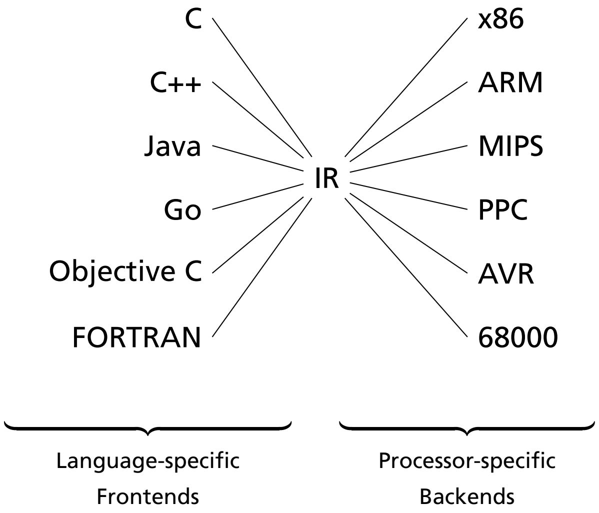
Portable Compilers

Building a compiler a large undertaking; most try to leverage it by making it portable.



Portable Compilers

Building a compiler a large undertaking; most try to leverage it by making it portable.



Intermediate Representations/Formats

Stack-Based IR: Java Bytecode

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
# javap -c Gcd

Method int gcd(int, int)
   0 goto 19

   3 iload_1          // Push a
   4 iload_2          // Push b
   5 if_icmple 15     // if a <= b goto 15

   8 iload_1          // Push a
   9 iload_2          // Push b
  10 isub             // a - b
  11 istore_1         // Store new a
  12 goto 19

  15 iload_2          // Push b
  16 iload_1          // Push a
  17 isub             // b - a
  18 istore_2         // Store new b

  19 iload_1          // Push a
  20 iload_2          // Push b
  21 if_icmpne 3      // if a != b goto 3

  24 iload_1          // Push a
  25 ireturn          // Return a
```

Stack-Based IRs

Advantages:

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers
- Often compact

Disadvantages:

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization

Register-Based IR: Mach SUIF

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
gcd:
gcd._gcdTmp0:
  sne   $vr1.s32 <- gcd.a,gcd.b
  seq   $vr0.s32 <- $vr1.s32,0
  btrue $vr0.s32,gcd._gcdTmp1   // if!(a != b) goto Tmp1

  sl    $vr3.s32 <- gcd.b,gcd.a
  seq   $vr2.s32 <- $vr3.s32,0
  btrue $vr2.s32,gcd._gcdTmp4   // if!(a < b) goto Tmp4

  mrk   2, 4     // Line number 4
  sub   $vr4.s32 <- gcd.a,gcd.b
  mov   gcd._gcdTmp2 <- $vr4.s32
  mov   gcd.a <- gcd._gcdTmp2   // a = a - b
  jmp   gcd._gcdTmp5
gcd._gcdTmp4:
  mrk   2, 6
  sub   $vr5.s32 <- gcd.b,gcd.a
  mov   gcd._gcdTmp3 <- $vr5.s32
  mov   gcd.b <- gcd._gcdTmp3   // b = b - a
gcd._gcdTmp5:
  jmp   gcd._gcdTmp0

gcd._gcdTmp1:
  mrk   2, 8
  ret   gcd.a    // Return a
```

Register-Based IRs

Most common type of IR

Advantages:

- Better representation for register machines
- Dataflow is usually clear

Disadvantages:

- Slightly harder to synthesize from code
- Less compact
- More complicated to interpret

Three-Address Code & Static Single Assignment

Most register-based IRs use **three-address code**: Arithmetic instructions have three operands: two sources and one destination.

SSA Form: each variable in an IR is assigned exactly once

C code:

```
int gcd(int a, int b)
{
  while (a != b)
    if (a < b)
      b -= a;
    else
      a -= b;
  return a;
}
```

Three-Address:

```
WHILE:  t = sne a, b
        bz DONE, t
        t = slt a, b
        bz ELSE, t
        b = sub b, a
        jmp LOOP
ELSE:   a = sub a, b
LOOP:   jmp WHILE
DONE:   ret a
```

SSA:

```
WHILE:  t1 = sne a1, b1
        bz DONE, t1
        t2 = slt a1, b1
        bz ELSE, t2
        b1 = sub b1, a1
        jmp LOOP
ELSE:   a1 = sub a1, b1
LOOP:   jmp WHILE
DONE:   ret a1
```

Basic Blocks

A **Basic Block** is a sequence of IR instructions with two properties:

1. The first instruction is the only entry point (no other branches in; can only start at the beginning)
2. Only the last instruction may affect control (no other branches out)

∴ If any instruction in a basic block runs, they all do

Typically "arithmetic and memory instructions, then branch"

```
ENTER:  t2 = add t1, 1
        t3 = slt t2, 10
        bz NEXT, t3
```

Basic Blocks and Control-Flow Graphs

```
WHILE:  t1 = sne a1, b1      ◀
        bz DONE, t1
        t2 = slt a1, b1      ◀
        bz ELSE, t2
        b1 = sub b1, a1      ◀
        jmp LOOP
ELSE:   a1 = sub a1, b1      ◀
LOOP:   jmp WHILE            ◀
DONE:   ret a1               ◀
```

- Leaders: branch targets & after conditional branch

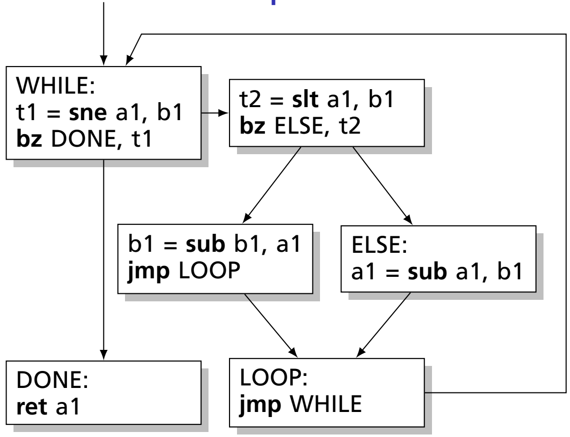
Basic Blocks and Control-Flow Graphs

```
WHILE:  t1 = sne a1, b1     ◀
        bz DONE, t1
-----------------------------
        t2 = slt  a1, b1    ◀
        bz ELSE, t2
-----------------------------
        b1 = sub b1, a1     ◀
        jmp LOOP
-----------------------------
ELSE:   a1 = sub a1, b1     ◀
-----------------------------
LOOP:   jmp WHILE           ◀
-----------------------------
DONE:   ret  a1             ◀
```

- Leaders: branch targets & after conditional branch
- Basic blocks: start at a leader; end before next

Basic Blocks and Control-Flow Graphs

```
WHILE:  t1 = sne a1, b1
        bz DONE, t1
-----------------------
        t2 = slt a1, b1
        bz ELSE, t2
-----------------------
        b1 = sub b1, a1
        jmp LOOP
-----------------------
ELSE:   a1 = sub a1, b1
-----------------------
LOOP:   jmp WHILE
-----------------------
DONE:   ret a1
```



- Leaders: branch targets & after conditional branch
- Basic blocks: start at a leader; end before next
- Basic Blocks are nodes of the Control-Flow Graph

The LLVM IR

Three-address code instructions; Static single-assignment; Explicit control-flow graph; Local names start with %; Types throughout; User-defined functions

```
int add(int x, int y)
{
  return x + y;
}
```

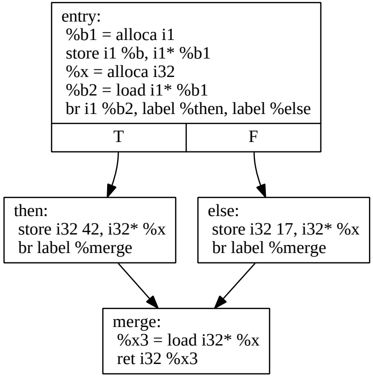
```
define i32 @add(i32 %x, i32 %y) {
entry:
  %x1 = alloca i32
  store i32 %x, i32* %x1
  %y2 = alloca i32
  store i32 %y, i32* %y2
  %x3 = load i32* %x1
  %y4 = load i32* %y2
  %tmp = add i32 %x3, %y4
  ret i32 %tmp
}
```

i32: 32-bit signed integer type
alloca: Allocate space on the stack; return a pointer
store: Write a value to an address
load: Read a value from an address
add: Add two values to produce a third
ret: Return a value to the caller

Basic Blocks

An LLVM function: a control-flow graph of basic blocks.

```
int cond(bool b) {
  int x;
  if (b) x = 42;
  else   x = 17;
  return x;
}
```



```
entry:
 %b1 = alloca i1
 store i1 %b, i1* %b1
 %x = alloca i32
 %b2 = load i1* %b1
 br i1 %b2, label %then, label %else
```

```
then:
 store i32 42, i32* %x
 br label %merge
```

```
else:
 store i32 17, i32* %x
 br label %merge
```

```
merge:
 %x3 = load i32* %x
 ret i32 %x3
```

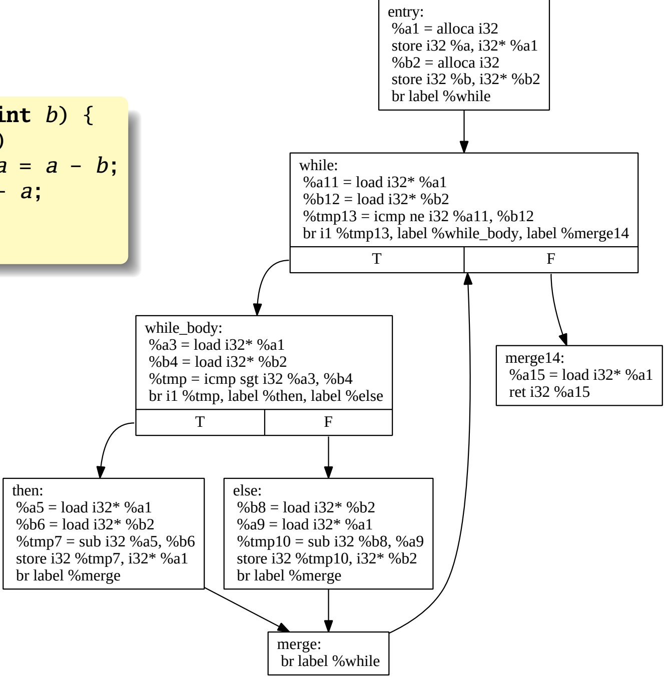
CFG for 'cond' function

```
define i32 @cond(i1 %b) {
entry:
  %b1 = alloca i1
  store i1 %b, i1* %b1
  %x = alloca i32
  %b2 = load i1* %b1
  br i1 %b2, label %then, label %else

merge:        ; preds = %else, %then
  %x3 = load i32* %x
  ret i32 %x3

then:         ; preds = %entry
  store i32 42, i32* %x
  br label %merge

else:         ; preds = %entry
  store i32 17, i32* %x
  br label %merge
}
```

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}
```

```
define i32 @gcd(i32 %a, i32 %b) {
entry:
  %a1 = alloca i32
  store i32 %a, i32* %a1
  %b2 = alloca i32
  store i32 %b, i32* %b2
  br label %while
while:                      ; preds = %merge, %entry
  %a11 = load i32* %a1
  %b12 = load i32* %b2
  %tmp13 = icmp ne i32 %a11, %b12
  br i1 %tmp13, label %while_body, label %merge14
while_body:                 ; preds = %while
  %a3 = load i32* %a1
  %b4 = load i32* %b2
  %tmp = icmp sgt i32 %a3, %b4
  br i1 %tmp, label %then, label %else
merge:                      ; preds = %else, %then
  br label %while
then:                       ; preds = %while_body
  %a5 = load i32* %a1
  %b6 = load i32* %b2
  %tmp7 = sub i32 %a5, %b6
  store i32 %tmp7, i32* %a1
  br label %merge
else:                       ; preds = %while_body
  %b8 = load i32* %b2
  %a9 = load i32* %a1
  %tmp10 = sub i32 %b8, %a9
  store i32 %tmp10, i32* %b2
  br label %merge
merge14:                    ; preds = %while
  %a15 = load i32* %a1
  ret i32 %a15
}
```

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}
```



```
entry:
 %a1 = alloca i32
 store i32 %a, i32* %a1
 %b2 = alloca i32
 store i32 %b, i32* %b2
 br label %while

while:
 %a11 = load i32* %a1
 %b12 = load i32* %b2
 %tmp13 = icmp ne i32 %a11, %b12
 br i1 %tmp13, label %while_body, label %merge14

while_body:
 %a3 = load i32* %a1
 %b4 = load i32* %b2
 %tmp = icmp sgt i32 %a3, %b4
 br i1 %tmp, label %then, label %else

then:
 %a5 = load i32* %a1
 %b6 = load i32* %b2
 %tmp7 = sub i32 %a5, %b6
 store i32 %tmp7, i32* %a1
 br label %merge

else:
 %b8 = load i32* %b2
 %a9 = load i32* %a1
 %tmp10 = sub i32 %b8, %a9
 store i32 %tmp10, i32* %b2
 br label %merge

merge:
 br label %while

merge14:
 %a15 = load i32* %a1
 ret i32 %a15
```

CFG for 'gcd' function

The MicroC Code Generator

Assumes AST is semantically correct
Translate each AST node into LLVM IR
Construct expressions bottom-up
Construct basic blocks for control-flow statements

`http://llvm.org`
`http://llvm.org/docs/tutorial`

`http://llvm.moe` Ocaml bindings documentation

The Code Generator (codegen.ml)

The *translate* function takes a semantically checked AST and returns an LLVM module

```
module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "MicroC" in

  (* ... *)
  let build_function_body fdecl =
  (* ... *)
  in
  List.iter build_function_body functions;
  the_module
```

The LLVM Type of Types

MicroC only supports primitive types; this could get complicated.

```
(* Get types from the context *)
let i32_t      = L.i32_type    context
and i8_t       = L.i8_type     context
and i1_t       = L.i1_type     context
and float_t    = L.double_type context
and void_t     = L.void_type   context in

(* Return the LLVM type for a MicroC type *)
let ltype_of_typ = function
    A.Int   -> i32_t
  | A.Bool  -> i1_t
  | A.Float -> float_t
  | A.Void  -> void_t
in
```

Define Global Variables

```
int i;
bool b;
int k;

int main()
{
  i = 42;
  k = 10;
```

```
@k  = global i32 0
@b  = global i1 false
@i  = global i32 0

define i32 @main() {
entry:
  store i32 42, i32* @i
  store i32 10, i32* @k
```

```
(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
  let global_var m (t, n) =
    let init = match t with
        A.Float -> L.const_float (ltype_of_typ t) 0.0
      | _ -> L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in
```

Declare external functions

Declare *printf*, which we'll use to implement various *print* functions and *printbig*, which illustrates linking with external C code

Formal function parameters are passed to LLVM in an OCaml array

```
let printf_t : L.lltype =
    L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
    L.declare_function "printf" printf_t the_module in

let printbig_t : L.lltype =
    L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
    L.declare_function "printbig" printbig_t the_module in
```

```
declare i32 @printf(i8*, ...)

declare i32 @printbig(i32)
```

Define function prototypes

```
void foo() ...

int bar(int a, bool b, int c) ...

int main() ...
```

```
define void @foo() ...

define i32 @bar(i32 %a, i1 %b, i32 %c)

define i32 @main() ...
```

Build a map from function name to (LLVM function, *fdecl*)

Construct the declarations first so we can call them when we build their bodies.

```
(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
  let function_decl m fdecl =
    let name = fdecl.sfname
    and formal_types = Array.of_list
        (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
    in let ftype =
      L.function_type (ltype_of_typ fdecl.styp) formal_types in
    StringMap.add name (L.define_function name ftype the_module,
                        fdecl) m in
  List.fold_left function_decl StringMap.empty functions in
```

build_function_body

An "Instruction Builder" is the LLVM library's object that controls where the next instruction will be inserted. It points to some instruction in some basic block.

This is an unfortunate artifact of LLVM being written in C++.

We also define string constants passed to *printf*.

```
(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) =
     StringMap.find fdecl.sfname function_decls in
  let builder =
     L.builder_at_end context (L.entry_block the_function) in

  let int_format_str =
     L.build_global_stringptr "%d\n" "fmt" builder
  and float_format_str =
     L.build_global_stringptr "%g\n" "fmt" builder in
```

```
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
```

Formals and Locals

Allocate formal arguments and local variables on the stack; remember names in *local_vars* map

```
int foo(int a, bool b)
{
  int c;
  bool d;
```

```
define i32 @foo(i32 %a, i1 %b) {
entry:
  %a1 = alloca i32
  store i32 %a, i32* %a1
  %b2 = alloca i1
  store i1 %b, i1* %b2
  %c = alloca i32
  %d = alloca i1
```

```
let local_vars =
  let add_formal m (t, n) p =
    L.set_value_name n p;
    let local = L.build_alloca (ltype_of_typ t) n builder in
    ignore (L.build_store p local builder);
    StringMap.add n local m
  and add_local m (t, n) =
        let local_var = L.build_alloca (ltype_of_typ t) n builder
        in StringMap.add n local_var m in

  let formals = List.fold_left2 add_formal StringMap.empty
    fdecl.sformals (Array.to_list (L.params the_function)) in
  List.fold_left add_local formals fdecl.slocals
```

lookup

Look for a variable among the locals/formal arguments, then the globals. Semantic checking ensures one of the two is always found.

Used for both identifiers and assignments.

```
(* Return the value for a variable or formal argument.
   Check local names first, then global names *)
let lookup n = try StringMap.find n local_vars
               with Not_found -> StringMap.find n global_vars
in
```

Expressions

The main expression function: build instructions in the given builder that evaluate an expression; return the expression's value

```
let rec expr builder ((_, e) : sexpr) = match e with
    SLiteral i     -> L.const_int i32_t i
  | SBoolLit b     -> L.const_int i1_t (if b then 1 else 0)
  | SFliteral l    -> L.const_float_of_string float_t l
  | SNoexpr        -> L.const_int i32_t 0
  | SId s          -> L.build_load (lookup s) s builder
  | SAssign (s, e) -> let e' = expr builder e in
                      ignore(L.build_store e' (lookup s) builder); e'
```

```
int a;

void foo(int c)
{
  a = c + 42;
}
```

```
@a = global i32 0

define void @foo(i32 %c) {
entry:
  %c1 = alloca i32
  store i32 %c, i32* %c1
  %c2 = load i32* %c1          ; read c
  %tmp = add i32 %c2, 42       ; tmp = c + 42
  store i32 %tmp, i32* @a      ; a = tmp
  ret void
}
```

Binary Operators: Floats

A trick: if the first operand is a float, treat it as a floating-point operation

```
| SBinop ((A.Float,_ ) as e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add     -> L.build_fadd
    | A.Sub     -> L.build_fsub
    | A.Mult    -> L.build_fmul
    | A.Div     -> L.build_fdiv
    | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
    | A.Neq     -> L.build_fcmp L.Fcmp.One
    | A.Less    -> L.build_fcmp L.Fcmp.Olt
    | A.Leq     -> L.build_fcmp L.Fcmp.Ole
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt
    | A.Geq     -> L.build_fcmp L.Fcmp.Oge
    | A.And | A.Or ->
    raise (Failure "internal error: semant should have rejected "
          ^ "and/or on float")
    ) e1' e2' "tmp" builder
```

Binary Operators: Integers

Evaluate left and right expressions; combine results

```
| SBinop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add     -> L.build_add
    | A.Sub     -> L.build_sub
    | A.Mult    -> L.build_mul
    | A.Div     -> L.build_sdiv
    | A.And     -> L.build_and
    | A.Or      -> L.build_or
    | A.Equal   -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less    -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
    ) e1' e2' "tmp" builder
```

neg/not/print/printb

Unary operators: evaluate subexpression and compute

```
| SUnop(op, ((t, _) as e)) ->
    let e' = expr builder e in
    (match op with
      A.Neg when t = A.Float -> L.build_fneg
    | A.Neg                  -> L.build_neg
    | A.Not                  -> L.build_not) e' "tmp" builder
```

Built-In Functions

Call external C functions that will be linked in later.

High-Level view of printbig.c:

```
#include <stdio.h> //Links in printf
void printbig(int c) {
  /* Code implementing printbig functionality */
}
```

print /*printb*: Invoke `printf("%d\n", v)`
printf: Invoke `printf("%g\n", v)`
printbig: Invoke `printbig(v)`

```
| SCall ("print", [e]) | SCall ("printb", [e]) ->
    L.build_call printf_func [| int_format_str ; (expr builder e) |]
                 "printf" builder
| SCall ("printbig", [e]) ->
    L.build_call printbig_func [| (expr builder e) |]
                 "printbig" builder
| SCall ("printf", [e]) ->
    L.build_call printf_func [| float_format_str ; (expr builder e) |]
                 "printf" builder
```

Function calls

Evaluate the actual arguments right-to-left and pass them to the call. *Do not name the result of void functions.*

```
| SCall (f, args) ->
   let (fdef, fdecl) = StringMap.find f function_decls in
   let llargs = List.rev (List.map (expr builder) (List.rev args)) in
   let result = (match fdecl.styp with
                    A.Void -> ""
                  | _ -> f ^ "_result") in
   L.build_call fdef (Array.of_list llargs) result builder
```

```
void foo(int a)
{
  print(a + 3);
}

int main()
{
  foo(40);
  return 0;
}
```

```
define void @foo(i32 %a) {
entry:
  %a1 = alloca i32
  store i32 %a, i32* %a1
  %a2 = load i32* %a1
  %tmp = add i32 %a2, 3
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr
    inbounds ([4 x i8]* @fmt1, i32 0, i32 0), i32 %tmp)
  ret void
}
define i32 @main() {
entry:
  call void @foo(i32 40)
  ret i32 0
}
```

Statements

Used to add a branch instruction to a basic block only of one doesn't already exist. Used by *if* and *while*

```
let add_terminal builder f =
  match L.block_terminator (L.insertion_block builder) with
    Some _ -> ()
  | None -> ignore (f builder) in
```

The main statement function: build instructions in the given builder for the statement; return the builder for where the next instruction should be placed. Semantic checking ensures *return* has an expression only in non-void functions

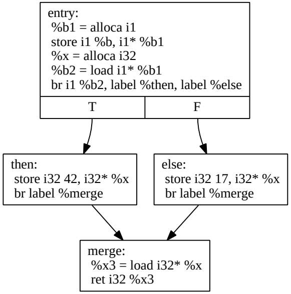
```
let rec stmt builder = function
    SBlock sl -> List.fold_left stmt builder sl
  | SExpr e   -> ignore(expr builder e); builder
  | SReturn e -> ignore(match fdecl.styp with
                          (* Special "return nothing" instr *)
                          A.Void -> L.build_ret_void builder
                          (* Build return statement *)
                        | _ -> L.build_ret (expr builder e) builder );
                builder
```

If Statements

Build basic blocks for *then*, *else*, and *merge*—where the next statement will be placed.

```
| SIf (predicate, then_stmt, else_stmt) ->
  let bool_val = expr builder predicate in
  let merge_bb = L.append_block context
                      "merge" the_function in
  let b_br_merge = L.build_br merge_bb  in

  let then_bb = L.append_block context
                       "then" the_function in
  add_terminal
   (stmt (L.builder_at_end context then_bb)
         then_stmt)
   b_br_merge;

  let else_bb = L.append_block context
                       "else" the_function in
  add_terminal
   (stmt (L.builder_at_end context else_bb)
         else_stmt)
   b_br_merge;

  ignore(L.build_cond_br bool_val then_bb
                         else_bb builder);
  L.builder_at_end context merge_bb
```

```
int cond(bool b) {
  int x;
  if (b) x = 42;
  else   x = 17;
  return x;
}
```


entry:
 %b1 = alloca i1
 store i1 %b, i1* %b1
 %x = alloca i32
 %b2 = load i1* %b1
 br i1 %b2, label %then, label %else

then:
 store i32 42, i32* %x
 br label %merge

else:
 store i32 17, i32* %x
 br label %merge

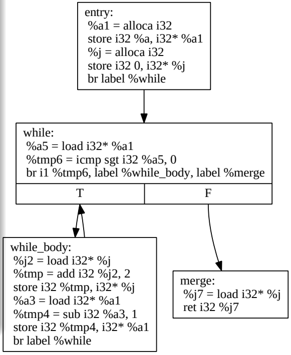
merge:
 %x3 = load i32* %x
 ret i32 %x3

CFG for 'cond' function

While Statements

```
| SWhile (predicate, body) ->
  let pred_bb = L.append_block context
                   "while" the_function in
  ignore(L.build_br pred_bb builder);

  let body_bb = L.append_block context
               "while_body" the_function in
  add_terminal
    (stmt (L.builder_at_end context body_bb)
          body)
    (L.build_br pred_bb);

  let pred_builder =
         L.builder_at_end context pred_bb in
  let bool_val =
              expr pred_builder predicate in

  let merge_bb = L.append_block context
                   "merge" the_function in
  ignore(L.build_cond_br bool_val
             body_bb merge_bb pred_builder);
  L.builder_at_end context merge_bb
```

```
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}
```



```
entry:
 %a1 = alloca i32
 store i32 %a, i32* %a1
 %j = alloca i32
 store i32 0, i32* %j
 br label %while
```

```
while:
 %a5 = load i32* %a1
 %tmp6 = icmp sgt i32 %a5, 0
 br i1 %tmp6, label %while_body, label %merge
```

```
while_body:
 %j2 = load i32* %j
 %tmp = add i32 %j2, 2
 store i32 %tmp, i32* %j
 %a3 = load i32* %a1
 %tmp4 = sub i32 %a3, 1
 store i32 %tmp4, i32* %a1
 br label %while
```

```
merge:
 %j7 = load i32* %j
 ret i32 %j7
```

CFG for 'foo' function

For Statements: Syntactic Sugar for While

```
for ( expr1 ; expr2 ; expr3 ) {
  body;
}
```

→

```
expr1;
while ( expr2 ) {
   body;
   expr3;
}
```

```
      | A.For (e1, e2, e3, body) -> stmt builder
            ( A.Block [A.Expr e1 ;
                       A.While (e2, A.Block [body ;
                                             A.Expr e3]) ] )
    in
```

The End

The remainder of *build_function_body*: build the body of the function by treating it as a block of statements; add a *return* if control fell off the end

```
    (* Build the code for each statement in the function *)
    let builder = stmt builder (SBlock fdecl.sbody) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.styp with
        A.Void -> L.build_ret_void
      | A.Float -> L.build_ret (L.const_float float_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
```

The body of *translate* (shown earlier): build the body of each function and return the module that was created.

```
  in
  List.iter build_function_body functions;
  the_module
```

The Top-Level

microc.ml (1/2)

Top-level of the MicroC compiler: handle command-line arguments

```
type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR),
                              "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist
      (fun filename -> channel := open_in filename) usage_msg;
```

microc.ml (2/2)

The actual compilation stuff: scan, parse, check the AST, generate LLVM IR, dump the module

```
let lexbuf = Lexing.from_channel !channel in

let ast = Microcparse.program Scanner.token lexbuf in

match !action with
  Ast -> print_string (Ast.string_of_program ast)
| _ -> let sast = Semant.check ast in
  match !action with
    Ast     -> ()

  | Sast    -> print_string (Sast.string_of_sprogram sast)

  | LLVM_IR -> print_string (Llvm.string_of_llmodule
                                   (Codegen.translate sast))

  | Compile -> let m = Codegen.translate sast in
      Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
```

Source Code Statistics

Source File	Lines	Role
scanner.mll	50	Token rules
microcparse.mly	115	Context-free grammar
ast.ml	106	Abstract syntax tree & pretty printer
sast.ml	77	Post-semantics AST
semant.ml	188	Semantic checking
codegen.ml	245	LLVM IR generation
microc.ml	32	Top-level
Total	813	

Test Case	Files	Total lines
Working .mc	42	539
Working outputs	42	334
Failing .mc	38	332
Error messages	38	38
Total	160	1243