Booleans and Control Structures

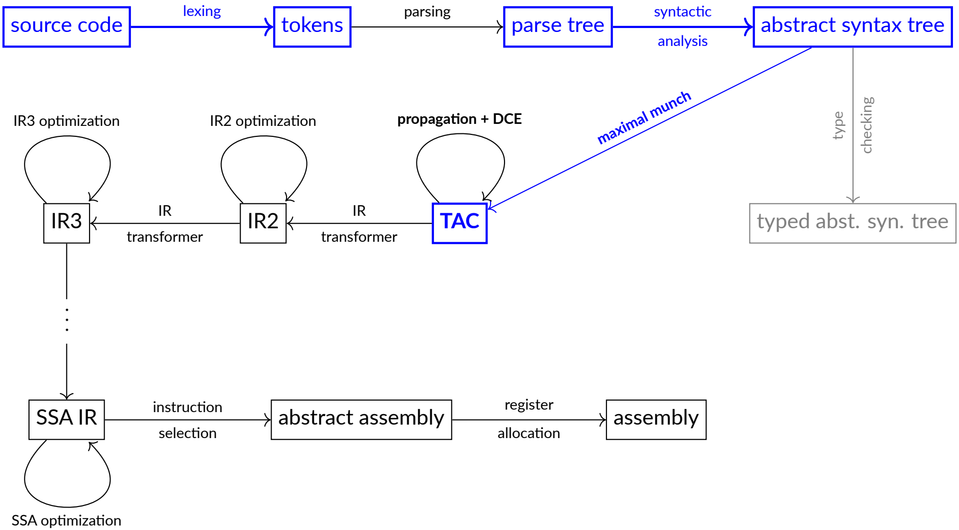
(CSC 3F002 EP) Compilers

Pierre-Yves Strub

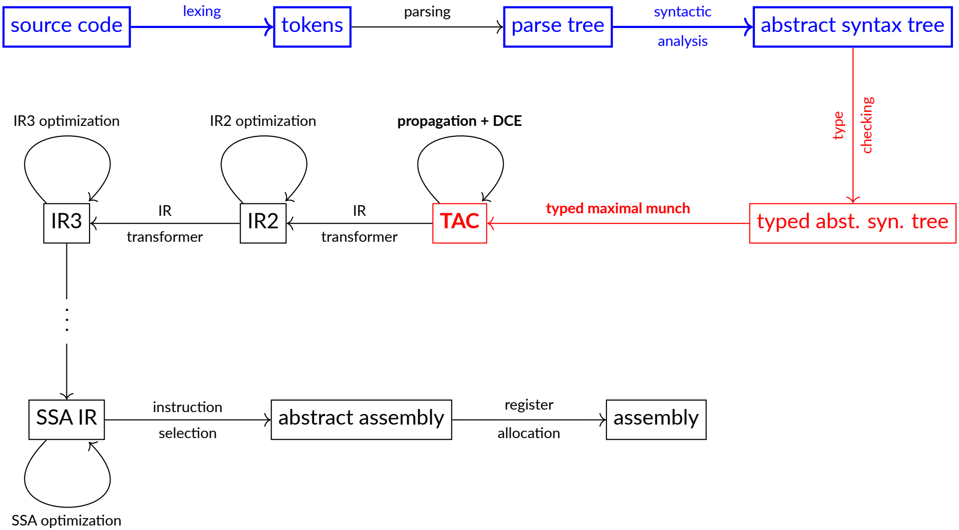
(Slide deck author: **Kaustuv Chaudhuri**)

2025-09-15 // Week 3

The Story So Far...



Today's focus



Today's Agenda

- Extending TAC with labels and jumps
  - Hand-written loops
  - Compiling BX boolean expressions
  - Short-circuiting boolean connectives
- Typing
  - Storing type information in the AST
- Control Structures
  - Conditionals
  - Loops
  - Structured jumps
- Local Declarations
  - From blocks to scopes
- *(Generalized Loops)*

Labels and Jumps

Extending TAC

- What we have in TAC so far:
  - Data transfer (`copy`, `const`)
  - Arithmetic (`add`, `sub`, ...)
  - Input/Output (`print`)
  - Linear instruction sequence — therefore finite
- Labels
  - A new kind of data, like temporaries
  - Labels can be used to name positions in the instruction sequence
  - Concrete syntax:
    - Global label: `[A-Za-z_][A-Za-z0-9_]*` (*think: BX identifiers*)
    - Local label: `.L(0|[1-9][0-9]*|[A-Za-z_][A-Za-z0-9_]*)`
- Control instructions
  - Local (intrafunction) jumps
  - Uses labels as *operands*
  - Function calls (*only basic, explained in lab 3 handout*)

TAC Labels and Jumps

(`%.L42` = example of local label and named temporary)

Instruction	Description
`%.L42:`	Label the position of the next instruction with `%.L42`. (This is not an instruction *per se* but an indication to the assembler about possible jumping destinations.)
`jmp %.L42;`	Unconditional jump to `%.L42`.
`jcc %1, %.L42;`	Conditional jump to `%.L42` depending on the value of `%1`, where `jcc` $\in$ {`jz`, `jnz`, `jl`, `jnl`, `jle`, `jnle`}.

jcc	condition	jcc	condition
`jz`	`%1 == 0`	`jnz`	`%1 != 0`
`jl`	`%1 < 0`	`jnl`	`%1 >= 0`
`jle`	`%1 <= 0`	`jnle`	`%1 > 0`

Looping Fibonacci

```
    %0 = const 0;
    %2 = const 1;
    print %0;
    %5 = add %2, %0;
    print %2;
    %6 = add %5, %2;
    print %5;
    %7 = add %6, %5;
    print %6;
    %8 = add %7, %6;
    print %7;
    %9 = add %8, %7;
    print %8;
    %10 = add %9, %8;
    print %9;
    %11 = add %10, %9;
    print %10;
    %12 = add %11, %10;
    print %11;
```

```
    %0 = const 0;
    %1 = const 1;
%.L1:
    print %0;
    %2 = add %0, %1;
    %0 = copy %1;
    %1 = copy %2;
    jmp %.L1
```

Looping + Terminating Fibonacci

```
    %0 = const 0;
    %1 = const 1;
    %2 = const 20;  // how many rounds to run
%.L1:
    jz %2, %.L2;
    print %0;
    %3 = add %0, %1;
    %0 = copy %1;
    %1 = copy %3;
    %4 = const 1;
    %2 = sub %2, %4;
    jmp %.L1;
%.L2:
```

Booleans and Comparisons in BX

(Also control structures, but that's next lecture)

- New primitive type: `bool`
- Expressions are now extended
  - Booleans constants: `true` and `false`
  - Comparisons between `int`s: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - Boolean connectives: `!`, `&&`, `||`
- Booleans in TAC:
  - TAC temporaries are still 64-bit signed integers (in 2's complement)
  - BX `false` is represented in TAC as the value `0`
    BX `true` is represented in TAC as any non-`0` value
  - Most of the time `true` will be represented as `1`, but allowing any non-`0` value gives us more freedom in translating BX to TAC

Lowering

- TAC can directly represent only comparisons with `0`.

jcc	condition	jcc	condition
`jz`	`%1 == 0`	`jnz`	`%1 != 0`
`jl`	`%1 < 0`	`jnl`	`%1 >= 0`
`jle`	`%1 <= 0`	`jnle`	`%1 > 0`

- Lowering: rewriting arbitrary BX comparison expressions in terms of these conditions

BX form	lowered
e_1 `==` e_2	`(`e_1 `-` e_2`) == 0`
e_1 `!=` e_2	`(`e_1 `-` e_2`) != 0`
e_1 `<` e_2	`(`e_1 `-` e_2`) < 0`
e_1 `<=` e_2	`(`e_1 `-` e_2`) <= 0`
e_1 `>` e_2	`(`e_1 `-` e_2`) > 0`
e_1 `>=` e_2	`(`e_1 `-` e_2`) >= 0`

BX	TAC
`true`	`1`
`false`	`0`
e_1 `==` e_2	`(`e_1 `-` e_2`)` (i.e., lhs of lowered ver.)

⋮

Boolean Connectives and Short-Circuiting

- The boolean connectives && and || are usually understood to be short-circuiting

```
false && e  =  false
true  || e  =  true
```

  In these cases, e is not computed.

- Short-circuiting ||:

```
def bmm(expr: Expression):
    # ...
    case BinopApp('||', left, right):
      t1, t2 = fresh_temp(), fresh_temp()
      emit(Instr(t1, 'const', 0, None))
      emit(Instr(t2, 'const', 0, None))
      l_after = fresh_label()
      emit(Instr(t1, 'copy', bmm(left), None))
      emit(Instr(None, 'jnz', t1, l_after))
      emit(Instr(t2, 'copy', bmm(right), None))
      emit(Instr(None, 'label', l_after, None))
      t = fresh_temp()
      emit(Instr(t, 'or', t1, t2))  # note: bitwise or
      return t
```

Short-Circuiting Example

BX

```
x = e₁ || e₂;
```

$\xrightarrow{\text{bmm}}$

TAC

```
    // %1, %2 are fresh
    %1 = const 0;
    %2 = const 0;
    // computation of ‘e₁’
    // with result in %10, say
    %1 = copy %10;
    // %.L1 is fresh
    jnz %1, %.L1;
    // computation of ‘e₂’
    // with result in %20, say
    %2 = copy %20;
%.L1:
    %3 = or %1, %2;
    // say ‘x’ is in %30
    %30 = copy %3;
```

Type-Directed Maximal Munch

Typed Abstract Syntax Tree

- Reminder: we are extending BX with booleans and boolean expressions
- Expressions can now have two possible types: `int` and `bool`
- Knowing the types of variables, every legal expression has a fixed type

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}} \qquad \frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 == e_2 : \texttt{bool}}$$

- It is common to precompute these types and store them in the expression nodes of the AST.

```
def type_check_expression(e: Expression):
  match e:
    case BinopApp(op, left , right):
      type_check_expression(left)
      type_check_expression(right)
      match op:
        case "+":
          check_type("int", left.ty)
          check_type("int", right.ty)
          e.ty = "int"
  # etc...
```

Typed Maximal Munch

- Until now, maximal munch has not used the type information (because it wasn't available in the parse trees!)
- With the typed AST (TAST), it is possible to use type information!
- There are many possible ways to use the type information
- Today: we will use types to munch boolean expressions specially
- Boolean Maximal Munch: (only showing top-down variant)

```
def tmm_bool_expr(bexpr, lab_true, lab_false):
    """Emit code to evaluate 'bexpr',
    jumping to 'lab_true' if true and
    jumping to 'lab_false' if false."""
```

Top-Down Boolean Maximal Munch

(a few illustrative cases – written in “Python”)

B	tmm_bool_expr(B, Lt, Lf)	proviso
true	emit(f”jmp {Lt};”)	—
false	emit(f”jmp {Lf};”)	—
e1 == e2	tmm_int_expr(e1, t1) tmm_int_expr(e2, t2) emit(f”{t1} = sub {t1}, {t2};”) emit(f”jz {t1}, {Lt};”) emit(f”jmp {Lf};”)	t1, t2 fresh
! B1	tmm_bool_expr(B1, Lf, Lt)	—
B1 && B2	tmm_bool_expr(B1, Li, Lf) emit(f”{Li}:”) tmm_bool_expr(B2, Lt, Lf)	Li fresh

Boolean Variables

(this is post lab 3)

- Using typed maximal munch, all boolean expressions turn into labels and jumps
- However, for boolean data, we need to produce numbers (esp. `0` and `1`)
- Example: boolean variables

```
var x = true : bool;
var y = false : bool;
var z = x || y : bool;
```

- Extending typed maximal munch for such assignments

S	tmm_stmt(S)	proviso

`x = B;`	`emit(f"{t} = const 0;")`
	`tmm_bool_expr(B, Lt, Lf)`
	`emit(f"{Lt}:")`
	`emit(f"{t} = const 1;")`
	`emit(f"{Lf}:")`
	`emit(f"{x} = copy {t};")`
		t, Lt, Lf fresh

Control Structures

Control Structures

- BX is a structured programming language
  - No arbitrary jumps
  - Instead, control moves through the program via particular programming constructs called control structures
- BX in lab 3 has the following control structures:
  - Conditionals: `if` ... `else` ...
  - Loops: `while` ...
  - Structured jumps: `break` and `continue`
- Later, BX will get:
  - Functions
  - Controlled exits: `return`
  - Finite loops: `for` ...

Conditionals

$\langle \text{stmt} \rangle$ ::= $\cdots$ | $\langle \text{ifelse} \rangle$

$\langle \text{ifelse} \rangle$ ::= $\texttt{"if"}$ $\texttt{"("}$ $\langle \text{expr} \rangle$ $\texttt{")"}$ $\langle \text{block} \rangle$ $\texttt{"else"}$ $\langle \text{block} \rangle$

$\langle \text{block} \rangle$ ::= $\texttt{"\{"}$ $\langle \text{stmt} \rangle^*$ $\texttt{"\}"}$

Conditionals

$\langle \text{stmt} \rangle ::= \cdots \mid \langle \text{ifelse} \rangle$

$\langle \text{ifelse} \rangle ::= \texttt{"if"}\ \texttt{"("}\ \langle \text{expr} \rangle\ \texttt{")"}\ \langle \text{block} \rangle\ \texttt{"else"}\ \langle \text{block} \rangle$

$\langle \text{block} \rangle ::= \texttt{"\{"}\ \langle \text{stmt} \rangle^*\ \texttt{"\}"}$

```
if (x == 1) {           // braces required
   print(x);
} else {                // else-clause required
   if (x % 2 != 0) {
      x = 3 * x + 1;
   } else { }           // else-clause requried
   x = x / 2;
}
```

Conditionals – optional `else`

$\langle\text{stmt}\rangle ::= \cdots \mid \langle\text{ifelse}\rangle$

$\langle\text{ifelse}\rangle ::= \texttt{"if"}\ \texttt{"("}\ \langle\text{expr}\rangle\ \texttt{")"}\ \langle\text{block}\rangle\ \langle\text{optelse}\rangle$
$\langle\text{optelse}\rangle ::= \epsilon \mid \texttt{"else"}\ \langle\text{block}\rangle$

$\langle\text{block}\rangle ::= \texttt{"\{"}\ \langle\text{stmt}\rangle^{*}\ \texttt{"\}"}$

```
if (x == 1) {        // braces required
   print(x);
} else {             // else-clause present
   if (x % 2 != 0) {
      x = 3 * x + 1;
   }                 // else-clause absent
   x = x / 2;
}
```

Conditionals – optional `else`

$\langle \text{stmt} \rangle ::= \cdots \mid \langle \text{ifelse} \rangle$

$\langle \text{ifelse} \rangle ::= \texttt{"if"}\ \texttt{"("}\ \langle \text{expr} \rangle\ \texttt{")"}\ \langle \text{block} \rangle\ \langle \text{optelse} \rangle$
$\langle \text{optelse} \rangle ::= \epsilon \mid \texttt{"else"}\ \langle \text{block} \rangle$

$\langle \text{block} \rangle ::= \texttt{"\{"}\ \langle \text{stmt} \rangle^*\ \texttt{"\}"}$

```
if (x == 1) {        // braces required
   print(x);
} else {             // else-clause present
   if (x % 2 != 0) {
      x = 3 * x + 1;
   }                 // else-clause absent
   x = x / 2;
}
```

```
if (x == 1) { print(1); }
else {
   if (x == 2) { print(2); }
   else {
      if (x == 3) { print(3); }
      // ...
   }
}
```

Conditionals – simplifying `else if`

$\langle\text{stmt}\rangle ::= \cdots \mid \langle\text{ifelse}\rangle$

$\langle\text{ifelse}\rangle ::= \texttt{"if"}\ \texttt{"("}\ \langle\text{expr}\rangle\ \texttt{")"}\ \langle\text{block}\rangle\ \langle\text{optelse}\rangle$
$\langle\text{optelse}\rangle ::= \epsilon \mid \texttt{"else"}\ \langle\text{block}\rangle \mid \texttt{"else"}\ \langle\text{ifelse}\rangle$

$\langle\text{block}\rangle ::= \texttt{"\{"}\ \langle\text{stmt}\rangle^{*}\ \texttt{"\}"}$

```
if (x == 1) { print(1); }
else {
  if (x == 2) { print(2); }
  else {
    if (x == 3) { print(3); }
    // ...
  }
}
```

vs.

```
if (x == 1)      { print(1); }
else if (x == 2) { print(2); }
else if (x == 3) { print(3); }
// ...
```

Munching Blocks and Conditionals

S	tmm_stmt(S)	proviso
{ S1 S2 ··· Sn }	tmm_stmt(S1) tmm_stmt(S2) ⋮ tmm_stmt(Sn)	—
if (cnd) Sthn else Sels	tmm_bool_expr(cnd, Lt, Lf) emit(f"{Lt}:") tmm_stmt(Sthn) emit(f"jmp {Lo};") emit(f"{Lf}:") tmm_stmt(Sels) emit(f"{Lo}:")	Lt, Lf, Lo fresh

`while` Loops

$\langle$stmt$\rangle$ ::= $\cdots$ | $\langle$while$\rangle$

$\langle$while$\rangle$ ::= `"while"` `"("` $\langle$expr$\rangle$ `")"` $\langle$block$\rangle$

S	tmm_stmt(S)	proviso

`while (Bc) Sbod`	`emit(f"{Lhead}:")`
	`tmm_bool_expr(Bc, Lbod, Lend)`
	`emit(f"{Lbod}:")`
	`tmm_stmt(Sbod)`
	`emit(f"jmp {Lhead};")`
	`emit(f"{Lend}:")`
		Lhead, Lbod, Lend fresh

Structured Jumps: `break`, `continue`

$\langle \text{stmt} \rangle ::= \cdots \mid \langle \text{jump} \rangle$

$\langle \text{jump} \rangle ::=$ `"break" ";"` $\mid$ `"continue" ";"`

- `break` jumps to the end of the innermost `while` loop (Lhead)
- `continue` jumps to the beginning of the innermost `while` loop (Lend)

Munching break, continue

```
# global variables used to illustrate -- do it differently in your own code
__break_stack = []
__continue_stack = []
```

S	tmm_stmt(S)	proviso
`break;`	`emit(f"jmp {__break_stack[-1]};")`	—
`continue;`	`emit(f"jmp {__continue_stack[-1]};")`	—

`while (Bc) Sbod`	`__break_stack.push(Lend)`
	`__continue_stack.push(Lhead)`
	`emit(f"{Lhead}:")`
	`tmm_bool_expr(Bc, Lbod, Lend)`
	`emit(f"{Lbod}:")`
	`tmm_stmt(Sbod)`
	`emit(f"jmp {Lhead};")`
	`emit(f"{Lend}:")`
	`__break_stack.pop()`
	`__continue_stack.pop()`
		Lhead, Lbod, Lend fresh labels

Local Declarations

Syntax of Variable Declarations

$\langle \text{type} \rangle$ ::= `"int"` | `"bool"`

$\langle \text{stmt} \rangle$ ::= $\cdots$ | $\langle \text{block} \rangle$ | $\langle \text{vardecl} \rangle$ *(blocks and declarations are statements)*

$\langle \text{vardecl} \rangle$ ::= `"var"` `IDENT` `"="` $\langle \text{expr} \rangle$ `":"` $\langle \text{type} \rangle$ `";"` *(initialization part of declaration)*

```
var x = 0 : int;                    // brings x into scope
var y = 1 : int;                    // brings y into scope
while (true) {
  var z = x + y : int;              // brings z into scope
  print(y);
  x = y;
  y = z;
  {
    var x = 42 : int;               // brings x into scope (shadowing outer x)
    print(x);                       // outputs 42
  }
  print(x);                         // outputs same as line 5
  // z goes out of scope
}
print(z);                           // ERROR: z not in scope
```

Type-Checking: Scope Management

```
scopes = [dict()]         # NB: in your own code, don’t use global variables

def find_variable_type(varname):
    for scope in reversed(scopes): # search scopes, most recent first
        # each scope maps a variable to its type
        if varname in scope:
            return scope[varname]
    raise ValueError(f’Variable {varname} not in scope’)

def type_check_stmt(s: Statement):
  match s:
    case Block(body):
      scopes.push(dict())   # create new scope
      for si in self.body:
        type_check_stmt(si)
      scopes.pop()          # at end of block, discard scope
```

Type-Checking: Variable Declarations

```
class VarDecl:
  name: str
  init: Expression
  ty:   str

def type_check_stmt(s: Statement):
  match s:
    case VarDecl(name, init, ty):
      if name in scopes[-1]:
        raise ValueError(f'Variable {self.name} already declared in same scope')
      type_check_expression(init)
      if init.ty != ty:
        raise ValueError('incompatible initializer type')
      scopes[-1][name] = ty
```

Type-Checking: Variable Declarations

```
class VarDecl:
  name: str
  init: Expression
  ty:   str

def type_check_stmt(s: Statement):
  match s:
    case VarDecl(name, init, ty):
      if name in scopes[-1]:
        raise ValueError(f'Variable {self.name} already declared in same scope')
      type_check_expression(init)
      if init.ty != ty:
        raise ValueError('incompatible initializer type')
      scopes[-1][name] = ty
```

- The mapping from a BX variable name to a TAC temporary can change depending on which scope the variable is in
- Therefore, you should make this mapping look something like `scopes`

Lab 3, week 1

- Week 1 tasks:
  - Extend the BX parser and type checker
  - Write a typed maximal munch, special casing boolean expressions
  - Write the munch cases for the control statements
- Write lots of tests. At least one test to trigger every path in your BX parser and your maximal munch implementation

Generalized Loops

`for` loops

- BX currently has only unbounded `while` loops, so `continue` is not very useful
- Many languages have bounded or ranged iteration loops, usually written with `for`, `foreach`, etc.

```
for (i in 1 .. 11) {
  if (i % 3 == 0) {
    continue;
  }
  print(i);
  if (i % 5 == 0) {
    break;
  }
}
```

Generalized Loops

- Generalized loop: common form for all looping constructs: `for`, `while`, …
- Generalized loop consists of several labeled code fragments:
  - Header: sets up and initializes loop vars
  - Body: main computation of the loop
  - Condition: thing that must be true for every run of the body
  - Footer: updates loop variables for next run
  - End: any computations to be done after loop end
- `break` and `continue` can be compiled to jumps to *end* and *footer*, respectively

Generalized Loop: Example

(Using an internal language that generalizes BX with unstructured jumps)

```
for (i in 1 .. 11) {
  if (i % 3 == 0) {
    continue;
  }
  print(i);
  if (i % 5 == 0) {
    break;
  }
}
```

⇒

```
{
header:
  var i = 1 : int;
condition:
  if (i >= 11) { goto end; }
body:
  {
    if (i % 3 == 0) {
      // continue
      goto footer;
    }
    print(i);
    if (i % 5 == 0) {
      // break
      goto end;
    }
  }
footer:
  i = i + 1;
  goto condition;
end:
}
```