CS406: Compilers

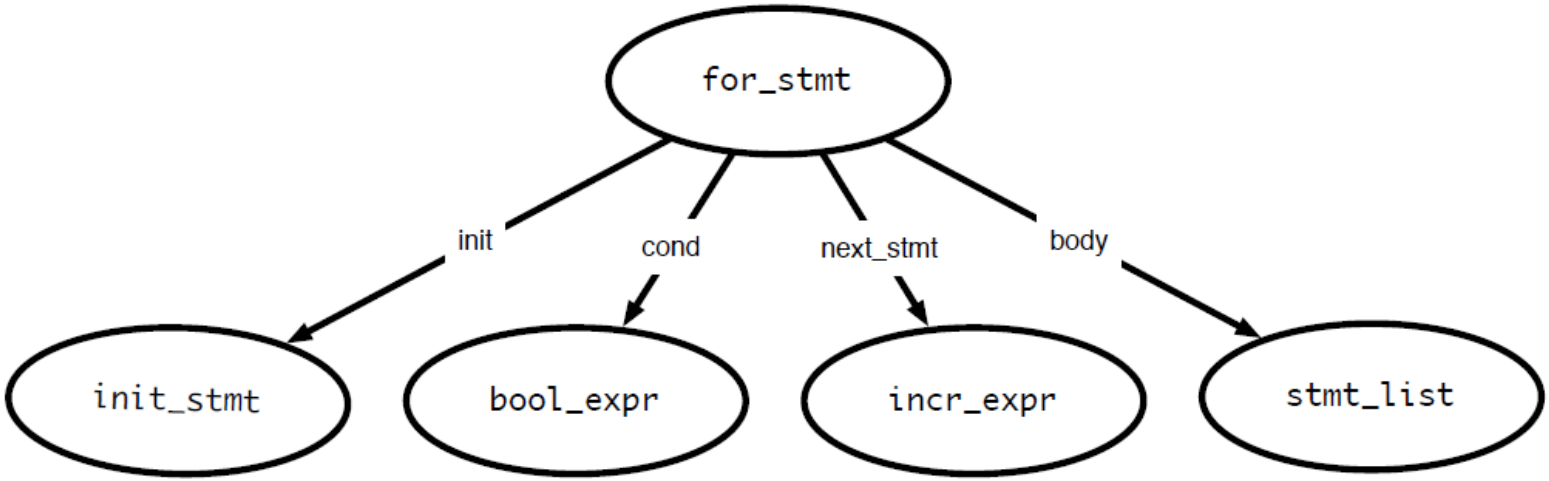
Spring 2020

Week 7: (IR) Code Generation - For Loops, Switch Statements, and Functions

(Slides courtesy: Prof. Milind Kulkarni)

For loops

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```



Generating code: for loops

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```

```
  <init_stmt>
LOOP:
  <bool_expr>
  j<!op> OUT
  <stmt_list>
INCR:
  <incr_stmt>
  jmp LOOP
OUT:
```

- Execute init_stmt first
- Jump out of loop if bool_expr is false
- Execute incr_stmt after block, jump back to top of loop
- Question: Why do we have the INCR label?

To handle a `continue` statement

Switch statements

```
switch (<expr>)
  case <const_list>: <stmt_list>
  case <const_list>: <stmt_list>
  ...
  default: <stmt_list>
end
```

- Generated code should evaluate <expr> and make sure that some case matches the result
- Question: how to decide where to jump?

Deciding where to jump

- Problem: do not know *which label* to jump to until switch expression is evaluated
- Use a jump table: an array indexed by case values, contains address to jump to
  - If table is not full (i.e., some possible values are skipped), can point to a default clause
    - If default clause does not exist, this can point to error code
  - Problems
    - If table is sparse, wastes a lot of space
    - If many choices, table will be very large

Jump table example

Consider the code:
((xxxx) is address of code)

Case x is
(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(0198) When 3 stmts;
(1000) When 5 stmts;
(1050) Else stmts;

Table only has one
Unnecessary row
(for choice 4)

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
4	JUMP 1050
5	JUMP 1000

Jump table example

Consider the code:
((xxxx) Is address of code)

Case x is
(0010) When 0: stmts0
(0017) When 1: stmts1
(0192) When 2: stmts2
(0198) When 3 stmts3
(1000) When 987 stmts4
(1050) When others stmts5

Table only has 983 unnecessary rows. Doesn't appear to be the right thing to do! ***NOTE: table size is proportional to range of choice clauses, not number of clauses!***

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
4	JUMP 1050
. . .	JUMP 1050
986	JUMP 1050
987	JUMP 1000

Linear search example

Consider the code:
(xxxx) Is offset of local Code start from the Jump instruction

Case x is
(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(1050) When others stmts;

If there are a small number of choices, then do an in-line linear search. A straightforward way to do this is generate code analogous to an IF THEN ELSE.

If (x == 0) then stmts1;
Elseif (x = 1) then stmts2;
Elseif (x = 2) then stmts3;
Else stmts4;

O(n) time, n is the size of the table, for each jump.

Dealing with jump tables

```
switch (<expr>)
  case <const_list>: <stmt_list>
  case <const_list>: <stmt_list>
  ...
  default: <stmt_list>
end
```

```
    <expr>
    <code for jump table>
  LABEL0:
    <stmt_list>
  LABEL1:
    <stmt_list>
  ...
  DEFAULT:
    <stmt_list>
  OUT:
```

- Generate labels, code, then build jump table
  - Put jump table after generated code
- Why do we need the OUT label?
  - In case of `break` statements

Functions

Terms

```
void foo() {
  int a, b;
  ...
  bar(a, b);
}


void bar(int x, int y) {
   ...
}
```

- foo is the *caller*
- bar is the *callee*
- a, b are the *actual parameters* to bar
- x, y are the *formal parameters* of bar
- Shorthand:
  - argument = actual parameter
  - parameter = formal parameter

Different Kinds of Parameters

- Value
- Reference
- Result
- Value-Reference
- Read-only
- Call-by-Name

Value parameters

- "Call-by-value"
- Used in C, Java, default in C++
- Passes the value of an argument to the function
- Makes a copy of argument when function is called
- Advantages? Disadvantages?

Advantage: 'side-effect' free – caller can be sure that the argument is not modified by the callee

Disadvantage: Not efficient for larger sized arguments.

Value parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```

- What do the print statements print?
- Answer:

  print(x); //prints 1

  print(x); //prints 1

Reference parameters

- "Call-by-reference"
- Optional in Pascal (use "var" keyword) and C++ (use "&")
- Pass the *address* of the argument to the function
- If an argument is an expression, evaluate it, place it in memory and then pass the address of the memory location
- Advantages? Disadvantages?

Advantage: Efficiency – for larger sized arguments
Disadvantage: results in clumsy code at times (e.g. check for null pointers)

Reference parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int &y, int &z) {
    y = 2;
    z = 3;
    print(x);
    print(y);
}
```

- What do the print statements print?
- Answer:

  print(x); //prints 3

  print(x); //prints 3

  print(y); //prints 3!

Result Parameters

- To capture the return value of a function
- Copied at the end of function into arguments of the caller
- E.g. `output` ports in Verilog `module` definitions

Result Parameters

```
int x = 1
void main () {
   foo(x, x);
   print(x);
}

void foo(int y, result int z) {
   y = 2;
   z = 3;
   print(x);
}
```

- What do the following statements print?

- Answer:

```
print(x); //prints 3
print(x) //prints 1
```

Value-Result Parameters

- “Copy-in copy-out”
- Evaluate argument expression, copy to parameters
- After subroutine is done, copy values of parameters back into arguments
- Results are often similar to pass-by-reference, but there are some subtle situations where they are different

Value-Result Parameters

```
int x = 1
void main () {
   foo(x, x);
   print(x);
}

void foo(int y, value result int z) {
   y = 2;
   z = 3;
   print(x);
}
```

- What do the following statements print?

- Answer:

```
print(x); //prints 3
print(x) //prints 1
```

Read-only Parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- E.g. `const` parameter in C/C++
- Enforcing becomes tricky when in the presence of aliasing and control flow. E.g.

```
void foo(readonly int x, int y) {
int * p;
if (...) p = &x else p = &y
*p = 4
}
```

Call-by-name Parameters

- The arguments are passed to the function before evaluation
  - Usually, we evaluate the arguments before passing them
- Not used in many languages, but Haskell uses a variant

```
int x = 1
void main () {
        foo(x+2);
        print(x);
}

void foo(int y) {
        z = y + 3; //expands to z = x + 2 + 3
        print(z);
}
```

Call-by-name Parameters

- Why is this useful?
  - E.g. to analyze certain properties of a program/function – termination

```
void main () {
        foo(bar());
}

void foo(int y) {
        z = 3;
        if(z > 3)
                z = y + z;
}
```

  - Even if `bar` has an infinite loop, the program terminates.

Other considerations

- Scalars
  - For call by value, can pass the address of the actual parameter and copy the value into local storage within the procedure
    - Reduces size of caller code (why is this good?)
  - If scalar is a constrained type (e.g., a Pascal range type), must insert type check for return values
  - For machines with a lot of registers (e.g., MIPS), compilers will save a few registers for arguments and return types
    - Less need to manipulate stack

Other considerations

- Arrays
  - For efficiency reasons, arrays should be passed by reference (why?)
    - Java, C, C++ pass arrays by reference by default (technically, they pass a pointer to the array by value)
  - Pass in a fixed size dope vector as the actual parameter (not the whole array!)
  - Callee can copy array into local storage as needed

Dope vectors

- Remember: store additional information about an array
  - Where it is in memory
  - Size of array
  - # of dimensions
  - Storage order
- Can sometimes eliminate dope vectors with compile-time analysis

Strings

- Requires a descriptor
  - Like a dope vector, provides information about string
- May just need to pass a pointer (if string contains information about its length)
- May also need to pass information about length

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Sections: TODO
- Fisher and LeBlanc: Crafting a Compiler with C
  - Sections: TODO