Compiler construction in4303 – lecture 11

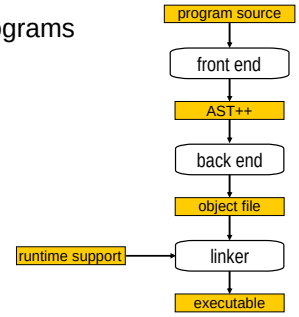
Imperative & OO languages: routines & control flow

Chapter 6.3 - 6.4

Overview

Imperative & OO programs

- context handling
  - …
  - object types
  - routine types
- code generation
  - control flow



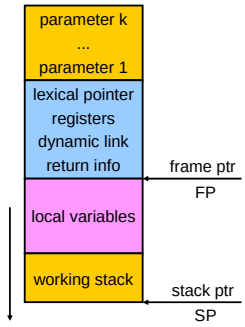
Routines and their activations

Functionality of a routine call

- supplying a new computing environment
- passing information to the new environment
- transfer of the flow-of-control to the new environment and (guaranteed) return
- returning info from the new environment

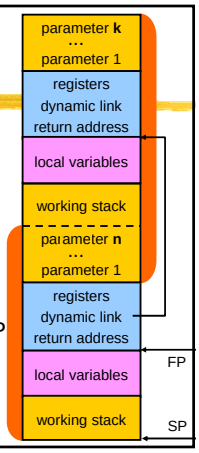
Activation records

- activation record holds
  - parameters
  - administration part
  - local variables
  - working stack
- routine activation order
  - last-in-first-out ⇒ stack
  - otherwise ⇒ heap



Routine invocation

- caller builds new stack frame for callee
  - push parameters
  - push admin
- transfer control to callee
  - stack return address, adjust FP
  - allocate space for local vars
  - execute code

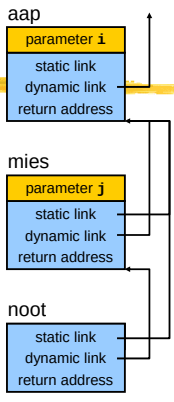


Nested routines

- activation record contains a lexical pointer or static link that points to outer scope

```
int aap( int i) {
   int noot() {return i+7;}
   int mies(int j) {return j+noot();}

   return mies(2)-3;
}
```



Exercise (5 min.)

- Given the GNU C routine:

```
void A(int a) {
   void B(int b) {
      void C(void) {
         printf("C called, a = %d\n", a);
      }
      if (b == 0) C() else B(b-1);
   }
   B(a);
}
```

a) draw the stack that results from the call A(2)

b) how does C access the parameter (a) from A?

Answers

Operations on routines

- calling
- passing as a parameter
- returning as a value
- currying [functional programming, chap 7]

Passing a routine as a parameter

flat

```
int aap(int i)                          {return i+7;}
int noot(int j, int (*f)(int))  {return (*f)(j);}
int mies(int k)                 {return noot(k,aap);}
```

address

nested

```
int noot(int (*f)())                {return (*f)();}
int mies(int i) {
  int aap()                             {return i+7;}

  return noot(aap);
}
```

static link

address

Returning a routine as a value

flat

```
typedef int (*fptr)(int);
int aap(int i) {return i+7;}
fptr mies()    {return aap;}
```

return routine address

nested

```
typedef int (*fptr)();
fptr mies(int i) {
  int aap()    {return i+7;}

  return aap;
}
```

return routine descriptor and allocate activation records on the heap

Lambda lifting

- remove static links, only global routines
- out-of-scope variables
  - referencing: pass additional pointers
  - creating: heap allocation

```
typedef int (*fptr)();
fptr mies(int i) {
  int aap()    {return i+7;}

  return aap;
}
```

nested

```
int aap(int *i) {return *i+7;}
fptr mies(int i) {
  int *_i = malloc(sizeof(i));
  *_i = i;

  return closure(aap,_i);
}
```

lifted

Code generation for control flow statements

- glue between basic blocks

transfer type	target instruction
jump	GOTO *label*
indirect jump	GOTO *register*
conditional jump	IF *a* ⊕ *b* THEN GOTO *label*

```
cmpl %eax,%edx
jle L3
```

Boolean expressions in flow of control

- why special code generation?
  - performance: avoid conversions
  - lazy semantics: || and && in C

```
if (list != NULL &&
    list->key < key) {
        ...
}
```

Code expr("`list != NULL`", r1)
Code expr("`list->key < key`", r2)
IF NOT(r1 and r2) THEN GOTO label-end
…
label-end:

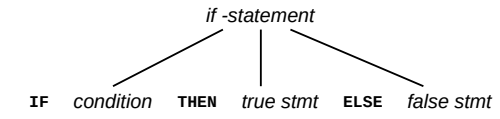
```
PROCEDURE Boolean control code(Node, True label, False label):
SELECT Node.type
  CASE Lazy and type:
    Boolean control code( Node.left, No label, False label);
    Boolean control code( Node.right, True label, False label);
  CASE Lazy or type:
    Boolean control code( Node.left, True label, No label);
    Boolean control code( Node.right, True label, False label);
  CASE Comparison type:
    Code expr( Node.left, R1)
    Code expr( Node.right, R2);
    IF True label /= No label:
      Emit("IF" R1 Node.op R2 "THEN GOTO" True label);
      IF False label /= No label:
        Emit( "GOTO" False label);
    ELSE
      Emit("IF" R1 Inv(Node.op) R2 "THEN GOTO" False label);
```

Falling through

```
PROCEDURE Boolean control code(Node, True label, False label):
SELECT Node.type
  CASE Lazy and type:
    IF False label /= No label:
      Boolean control code( Node.left, No label, False label);
      Boolean control code( Node.right, True label, False label);
    ELSE
      // The lazy_and should fall through on failure
      SET Fall through TO New label ();
      Boolean control code( Node.left, No label, Fall through);
      Boolean control code( Node.right, True label, No label);
      Emit("LABEL" Fall through ":");
```

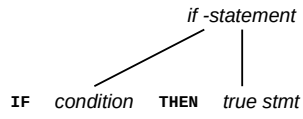
Error in the book, no errata yet

The if-statement




```
  Boolean control code( condition, No_label, false_label)
  Code statement( true stmt)
  GOTO end_label;
false_label:
  Code statement( false stmt)
end_label:
```

The if-statement




```
  Boolean control code( condition, No_label, end_label)
  Code statement( true stmt)
end_label:
```

The case-sta

CASE *expression* **IN**
  I_1 : *statement*$_1$
  ...
  I_n : *statement*$_n$
  ELSE *statement*$_{default}$
END CASE

Code expr(*expression*, R_{expr})
  IF R_{expr} = I_1 **THEN GOTO** label_1
  ...
  IF R_{expr} = I_n **THEN GOTO** label_n
  GOTO label_else;
label_1:
  Code statement (*statement*$_1$)
  GOTO end_label;
  ...
label_n:
  Code statement (*statement*$_n$)
  GOTO end_label;
label_else:
  Code statement (*statement*$_{default}$)
end_label:

The case-statement

CASE *expression* **IN**
  I_1 : *statement*$_1$
  ...
  I_n : *statement*$_n$
  ELSE *statement*$_{default}$
END CASE

jump table

```
.data
case_table:
        label_low
        ...
        label_high
```

Code expr(*expression*, R_{expr})
  IF R_{expr} < I_{low} **THEN GOTO** label_else
  IF R_{expr} > I_{high} **THEN GOTO** label_else
  GOTO case_table[R_{expr}- I_{low}];
label_1:
  ...

The repeat statement

repeat -statement

REPEAT *statement* **UNTIL** *condition* **END REPEAT**

repeat_label:
  Code statement(*statement*)
  Boolean control code(*condition*, No_label , repeat_label)

Exercise (3 min.)

while -statement

WHILE *condition* **DO** *statement* **END WHILE**

- give a translation schema for while statements

Answers

The for-statement

FOR i IN *lower bound* **..** *upper bound* **DO**
  statement
END FOR

  Code expr(*lower bound*, R_i)
  Code expr(*upper bound*, R_{ub})
  IF R_i > R_{ub} **THEN GOTO** end_label
loop_label:
  Code statement (*statement*)
  IF R_i = R_{ub} **THEN GOTO** end_label
  R_i := R_i+1;
  GOTO loop_label;
end_label:

test before increment