

## Compiler construction in4020 – lecture 11

Koen Langendoen

Delft University of Technology  
The Netherlands

## Overview

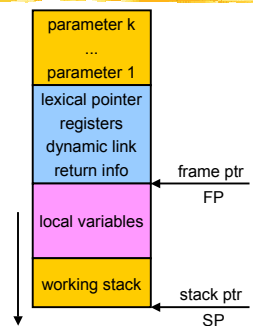
- routine types
  - activation records
  - invocation
  - values
- code generation for control flow statements
  - boolean expressions
  - selection statements
  - repetition statements

## 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

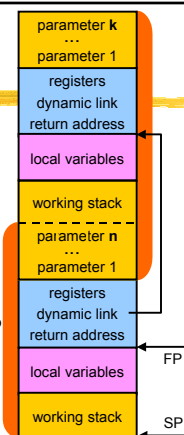
## Activation records

- activation record holds
  - local variables
  - parameters
  - working stack
  - administration part
- routine activation order
  - last-in-first-out  $\Rightarrow$  stack
  - otherwise  $\Rightarrow$  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

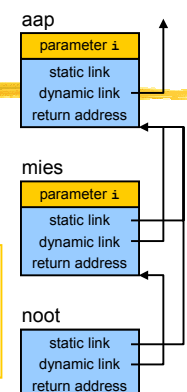


## Nested routines

- 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 i) {return i+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);
}
```

- draw the stack that results from the call A(2)
- 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 i, int (*f)(int)) {return (*f)(i);}
int mies(int i) {return noot(i,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

nested

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

lifted

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

## Code generation for control flow statements

- glue between basic blocks

transfer type	target instruction
jump	GOTO <i>label</i>
indirect jump	GOTO <i>register</i>
conditional jump	IF $a \oplus b$ THEN GOTO <i>label</i>

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

## Boolean expressions in flow of control

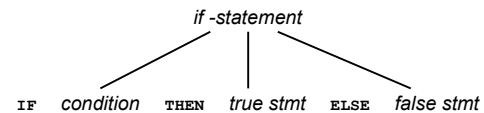
- why special code generation?
  - performance: avoid conversions
  - lazy semantics: `||` and `&&` in C

```
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);
```

## 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 case-statement

```
CASE expression IN
    I1 : statement1
    ...
    In : statementn
    ELSE statementdefault
END CASE
```

```
Code expr( expression, Rexpr)
IF Rexpr = I1 THEN GOTO label_1
...
IF Rexpr = In THEN GOTO label_n
GOTO label_else;
label_1:
    Code statement( statement1)
    GOTO end_label;
...
label_n:
    Code statement( statementn)
    GOTO end_label;
label_else:
    Code statement( statementdefault)
end_label:
```

## The case-statement

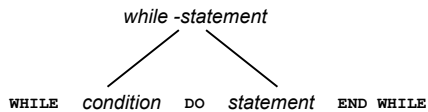
```
CASE expression IN
    I1 : statement1
    ...
    In : statementn
    ELSE statementdefault
END CASE
```

jump table

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

```
Code expr( expression, Rexpr)
IF Rexpr < Ilow THEN GOTO label_else
IF Rexpr > Ihigh THEN GOTO label_else
GOTO case_table[Rexpr - Ilow];
label_1:
    ...
```

## The while-statement



```

GOTO test_label;
loop_label:
  Code statement( statement)
test_label:
  Boolean control code( condition, loop_label , No_label)
  
```

## The for-statement

```

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

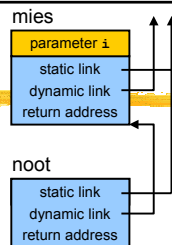
```

Code expr( lower bound, Ri)
Code expr( upper bound, Rub)
IF Ri > Rub THEN GOTO end_label
loop_label:
  Code statement( statement)
  IF Ri = Rub THEN GOTO end_label
  Ri := Ri + 1;
  GOTO loop_label;
end_label:
  
```

test before  
increment

## Summary

- (nested) routines
  - activation records
  - routine descriptors
- code generation for control flow statements
  - conditional expressions: true & false labels
  - case-statement: jump table
  - for-statement: overflow



## Homework

- study sections:
  - 6.5 Code generation for modules
- assignment 2:
  - make Asterix OO
  - deadline June 4 08:59
- print handout for next week [blackboard]