Compilers (Sheet #3)

Marius Gavrilescu

4.1
```
PROC double 0 0 0
LDLW 16
LDLW 16
PLUS
RETURNW

PROC apply3 0 0 0
CONST 3
CONST 0
LDLW 16
PCALLW 1
RETURNW

GLOBAL _double
CONST 0
GLOBAL _apply3
PCALLW 1
CONST 0
GLOBAL _print_num
PCALL 1
CONST 0
GLOBAL _newline
PCALL 0
```

4.2
```
proc flip(x^1: integer): integer;
  proc flop(y^2: integer): integer;
  begin
    if y^2 = 0 then return 1 else return flip(y^2-1) + x^1 end
  end;
begin
  if x^1 = 0 then return 1 else return 2 * flop(x^1-1) end
end;
```

flip4 calls flop3 calls flip2 calls flop1 calls flip0

```
              # begin activation record for flip(4)
4             # parameter
0             # static link
_flip         # CP
<return addr>
0             # dynamic link
              # ADDRESS: ADR1
```

```
                # begin activation record for flop(3)
    3           # parameter
    ADR1        # static link
    _flop       # CP
    <return addr>
    ADR1        # dynamic link
                # ADDRESS: ADR2

                # begin activation record for flip(2)
    2           # parameter
    0           # static link
    _flip       # CP
    <return addr>
    ADR2        # dynamic link
                # ADDRESS: ADR3

                # begin activation record for flop(1)
    1           # parameter
    ADR3        # static link
    _flop       # CP
    <return addr>
    ADR3        # dynamic link
                # ADDRESS: ADR4

                # activation record for flip(0)
    0           # parameter
    0           # static link
    _flip       # CP
    <return addr>
    ADR4        # dynamic link
```

4.3

```
                # activation record for fac(3, id)
    0           # parameter (static link of id)
    _id         # parameter
    3           # parameter
    0           # static link
    _fac        # CP
    <return addr>
    0           # dynamic link
                # ADDRESS: ADR1

                # activation record for fac(2, k1)
    ADR2        # parameter (static link of k1)
    _k1         # parameter
    2           # parameter
    0           # static link
    _fac        # CP
    <return addr>
    ADR1        # dynamic link
```

```
              # ADDRESS: ADR2

              # activation record for fac(1, k1)
ADR3          # parameter (static link of k1)
_k1           # parameter
1             # parameter
0             # static link
_fac          # CP
<return addr>
ADR2          # dynamic link
              # ADDRESS: ADR3

              # activation record for fac(0, k1)
_k1           # parameter
0             # parameter
0             # static link
_fac          # CP
<return addr>
ADR3          # dynamic link
              # ADDRESS: ADR4

              # activation record for k1(1)
1             # parameter
ADR3          # static link
_k1           # CP
<return addr>
ADR4          # dynamic link
              # ADDRESS: ADR5

              # activation record for k1(1)
1             # parameter
ADR2          # static link
_k1           # CP
<return addr>
ADR5          # dynamic link
              # ADDRESS: ADR6

              # activation record for k1(2)
2             # parameter
ADR1          # static link
_k1           # CP
<return addr>
ADR6          # dynamic link
              # ADDRESS: ADR7

              # activation record for id(6)
6             # parameter
0             # static link
_id           # CP
<return addr>
```

```
    ADR7          # dynamic link
                  # ADDRESS: ADR8
```

4.4

```
                  # activation record for sum(a)
    _a            # parameter
    0             # static link
    _sum          # CP
    <return addr>
    0             # dynamic link
                  # ADDRESS: ADR1

                  # activation record for doVec(add, v)
    _a            # parameter
    ADR2          # parameter (static link of add)
    _add          # parameter
    0             # static link
    _doVec        # CP
    <return addr>
    ADR1          # dynamic link
                  # ADDRESS: ADR2

                  # activation record for add(1)
    1             # parameter
    ADR2          # static link
    _add          # CP
    <return addr>
    ADR2          # dynamic link
                  # ADDRESS: ADR3


    ! f(v[i])
    GLOBAL _v     ! pushes v
    LDLW -4       ! pushes i
    OFFSET        ! pops twice, pushes v+i (that is, &v[i])
    LOADW         ! pops once, pushes v[i]
    LDLW 16       ! pushes _add
    LDLW 20       ! pushes the static link of _add
    PCALL 1       ! calls _add with the given static link and argument

    ! s := s + x
    LDLW 12       ! pushes the static link
    CONST -4      ! pushes offset of s from static link
    OFFSET        ! pops twice, pushes address of s
    LOADW         ! pops once, pushes s
    LDLW 16       ! pushes x
    PLUS          ! pops twice, pushes s+x
    LDLW 12       ! pushes the static link
    CONST -4      ! pushes offset of s from static link
    OFFSET        ! pops twice, pushes address of s
    STOREW        ! pops twice, stores the s+x at the address of s
```

```
! doVec(add, v)
LDLW 16       ! pushes v
LOCAL 0       ! pushes the base pointer
              ! (to be used as static link of _add)
GLOBAL _add   ! pushes _add
GLOBAL _doVec ! pushes _doVec
LOCAL 0       ! pushes the base pointer
              ! (to be used as static link of _doVec)
PCALL 3
```

If the parameter v was passed by value then as discussed in the lecture we would pretend it was passed by reference and modify the prelude of the functions to copy the contents of the array into the stack frame, which is an expensive operation.

Pass-by-value might be faster in the degenerate case of a one-element array, but if we use the method described above it would still be slower than passing the single element as a parameter.

4.5 Some architectures have alignment requirements, which means certain types can only be stored at certain addresses (for example, double words on i386 can only be stored at addresses that are multiples of 4 bytes). A compiler needs to take this into account when compiling records, because the record might require padding to ensure alignment.

The types `integer` and `array 4 of char` both have size 4, but the alignment for `integer` on i386 is 8 while the alignment for `array 4 of char` is 1.

The type rec is probably represented as one byte for c1, one byte for c2, two bytes of padding, and then 4 bytes for n.

Assuming g is called by the main function:

```
              # activation record for g()
0             # static link
_g            # CP
<return addr>
0             # dynamic link
              # ADDRESS: ADR1

              # activation record for f(s)
ADR1-4        # parameter
0             # static link
_f            # CP
<return addr>
ADR1          # dynamic link
              # ADDRESS: ADR2

! r.n := r.n + 1
LDLW 16       ! load first argument (address of r)
CONST 4
OFFSET        ! compute address of r.n
LOADW         ! load r.n
CONST 1       ! load 1
PLUS          ! compute r.n + 1
LDLW 16       ! load first argument (address of r)
```

```
CONST 4
OFFSET        ! compute address of r.n
STOREW        ! store r.n + 1 at address of r.n
```

For the Java-like language the code for the assignment should be identical, because previously we were passing a record by reference and now we are passing a pointer to the record by value, so the same number of dereferencings is needed.

We can show parameters are passed by value by modifying them and checking if the modification took place.

```
var a : rec;
var b : rec;

proc changeIt(x : rec);
begin
  x := b;
end;

proc test();
begin
  a.n := 0;
  b.n := 1;
  changeIt(a);
end;
```

If at the end of the test subroutine `a.n == 1`, then we know the parameter was passed by reference (because a was replaced with b). If it is still 0 then we know the parameter was passed by value (because the assignment in changeIt had no effect outside the function).