



Compilers

Activations

- Two goals:
 - Correctness
 - Speed
- Complications in code generation come from trying to be fast as well as correct

Two assumptions:

1. Execution is sequential; control moves from one point in a program to another in a well-defined order
2. When a procedure is called, control always returns to the point immediately after the call

- An invocation of procedure **P** is an *activation* of **P**
- The *lifetime* of an activation of **P** is
 - All the steps to execute **P**
 - Including all the steps in procedures **P** calls

- The *lifetime* of a variable **x** is the portion of execution in which **x** is defined
- Note that
 - Lifetime is a dynamic (run-time) concept
 - Scope is a static concept

- Observation
 - When **P** calls **Q**, then **Q** returns before **P** returns
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

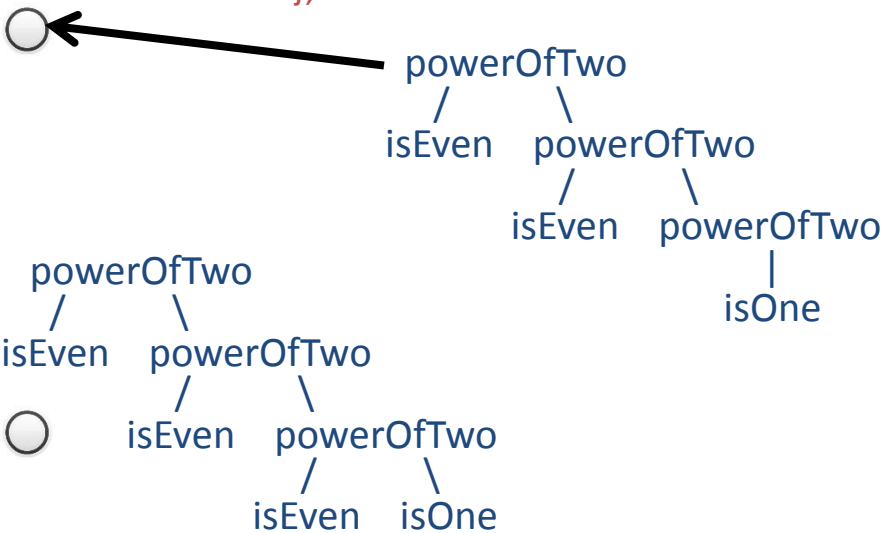
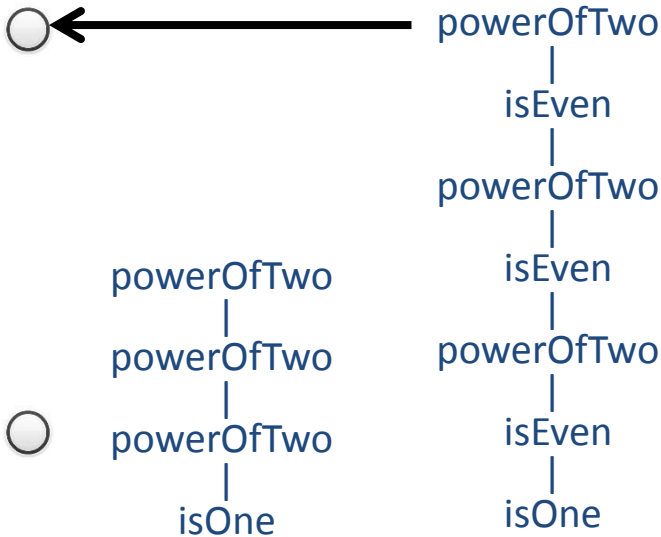
```
Class Main {  
    g() : Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};  
}
```

```
Class Main {  
    g() : Int { 1 };  
    f(x:Int): Int { if x = 0 then g() else f(x - 1) fi};  
    main(): Int {{f(3); }};  
}
```


The `powerOfTwo()` function, shown to the right, returns true if its argument is a power of two, false otherwise. What is the activation tree for `powerOfTwo(4)`?

Activations

```
isEven(x:Int) : Bool { x % 2 == 0 };
isOne(x:Int) : Bool { x == 1 };
powerOfTwo(x:Int) : Bool {
  if isEven(x) then powerOfTwo(x / 2)
  else isOne(x)
};
```



- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

```
Class Main {  
    g() : Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};  
}
```

Memory

