Code Generation II

FEBRUARY 6, 2014

3-Address Code

Abstraction of assembly code.

Similar enough to allow certain optimizations.

- E.g. `push r0; pop r0` can be dropped

Abstract enough to target different hardware.

- E.g. gcc uses the same front-end for all target platforms

3-Address Code

```
x := y ⊙ z
x := ⊙ y
x := y
x[i] := y
x := y[i]
if x ⊙ y goto z
goto x
```

Up to ***three addresses*** per statement.

Addresses may store ***operands*** or ***results***.

“Addresses” may be ***constants***, ***registers***, ***symbol names***, or ***labels***.

Code Generation for Expressions

```
expr ::= ID <- expr
       | expr[@TYPE].ID( [ expr [[, expr]]* ] )
       | ID( [ expr [[, expr]]* ] )
       | if expr then expr else expr fi
       | while expr loop expr pool
       | { [[expr; ]]+ }
       | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]]* in expr
       | case expr of [[ID : TYPE => expr; ]]+ esac
       | new TYPE
       | isvoid expr
       | expr + expr
       | expr - expr
       | expr * expr
       | expr / expr
       | ~expr
       | expr < expr
       | expr <= expr
       | expr = expr
       | not expr
       | (expr)
       | ID
       | integer
       | string
       | true
       | false
```

Code Generation For Expressions

```
expr ::= ID <- expr
       | expr[@TYPE].ID( [ expr [, expr]* ] )
       | ID( [ expr [, expr]* ] )
       | if expr then expr else expr fi
       | while expr loop expr pool
       | { [expr; ]+ }
       | let ID : TYPE [ <- expr ] [, ID : TYPE [ <- expr ]]* in expr
       | case expr of [ID : TYPE => expr; ]+ esac
       | new TYPE
       | isvoid expr
       | expr + expr
       | expr - expr
       | expr * expr
       | expr / expr
       | ~ expr
       | expr < expr
       | expr <= expr
       | expr = expr
       | not expr
       | (expr)
       | ID
       | integer
       | string
       | true
       | false
```

80% Complete!

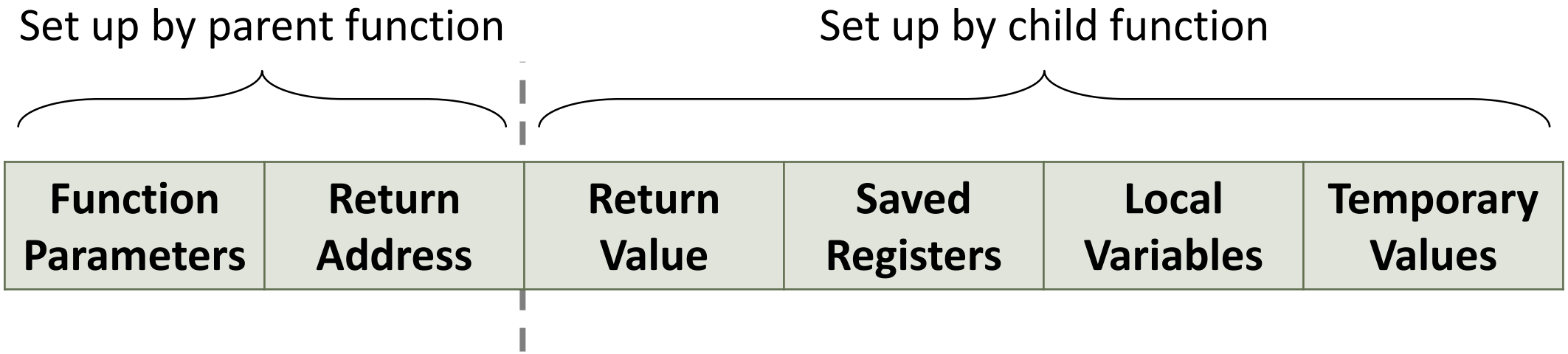
Function Calls

```
expr ::= ID <- expr
       | expr[@TYPE].ID( [ expr [[, expr]]* ] )
       | ID( [ expr [[, expr]]* ] )
       | if expr then expr else expr fi
       | while expr loop expr pool
       | { [[expr; ]]+ }
       | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]]* in expr
       | case expr of [[ID : TYPE => expr; ]]+ esac
       | new TYPE
       | isvoid expr
       | expr + expr
       | expr - expr
       | expr * expr
       | expr / expr
       | ~expr
       | expr < expr
       | expr <= expr
       | expr = expr
       | not expr
       | (expr)
       | ID
       | integer
       | string
       | true
       | false
```

Activation Records

Memory allocation for a single function call.

Also known as ***call frames*** or ***stack frames***.



Activation Records

Function Parameters	Return Address	Return Value	Saved Registers	Local Variables	Temporary Values

The ***frame pointer*** identifies the start of the record.

- Typically set by callee based on stack pointer.

Some fields may be kept in registers.

- Cool: `ra` register for return addresses.
- x86_64 keeps (some) parameters and return value in registers.

Calling Conventions

Pre- and ***post-conditions*** for function calls.

- Specify how arguments are passed.
- Specify how to return the result.
- Specify which registers are unaffected by call.

Which Calling Conventions to Use?

COOL COMPILERS

- Cool's `call` instruction sets `ra`.
- Otherwise, it's entirely up to you.

X86_64 COMPILERS

- x86_64's `call` instruction stores address on stack.
- Must be consistent to call external functions (e.g. `puts`).
- Refer to x86_64 Machine Level Programming.

Function Call Example

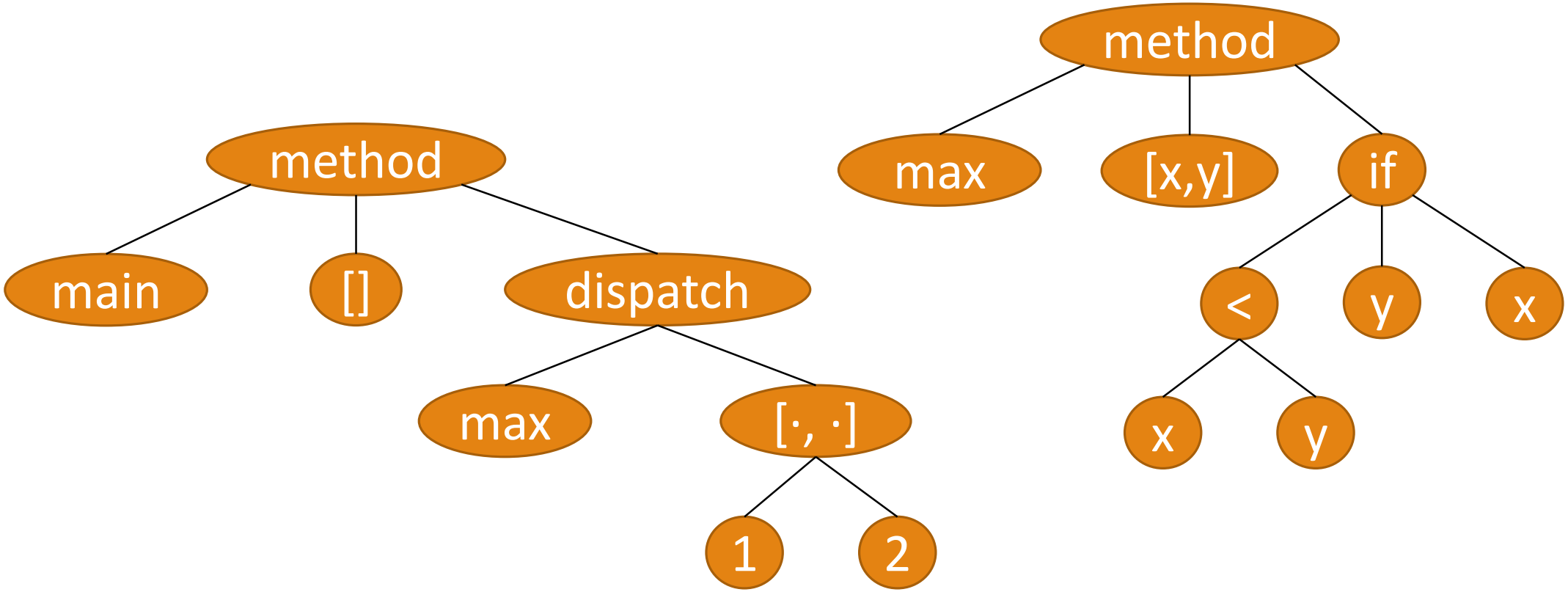
Cool virtual machine.

Calling conventions:

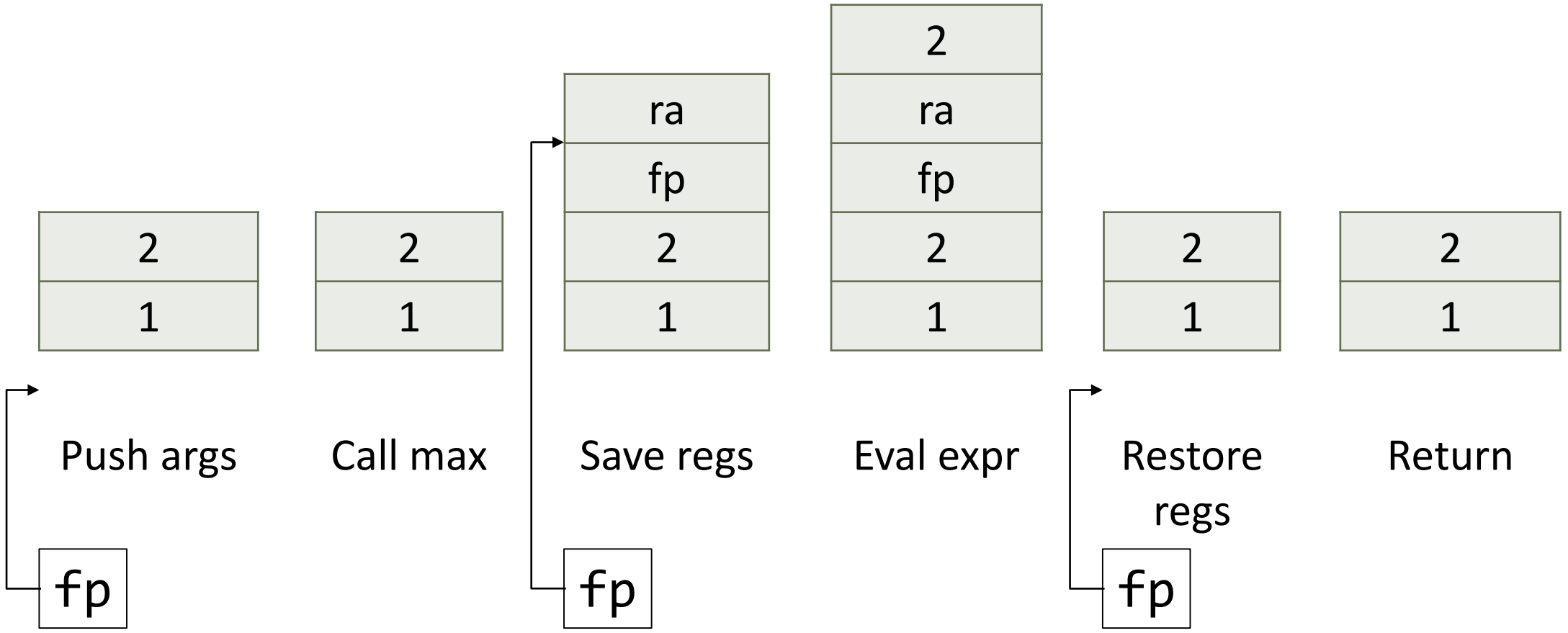
- Arguments are passed on stack. Arg 1 is below arg N.
- Return value in `r1`.
- All other registers are callee-saved.

```
max(Int x, Int y) : Int {
  if (x < y) then y else x fi
}
main() : Object {
  max(1,2)
}
```

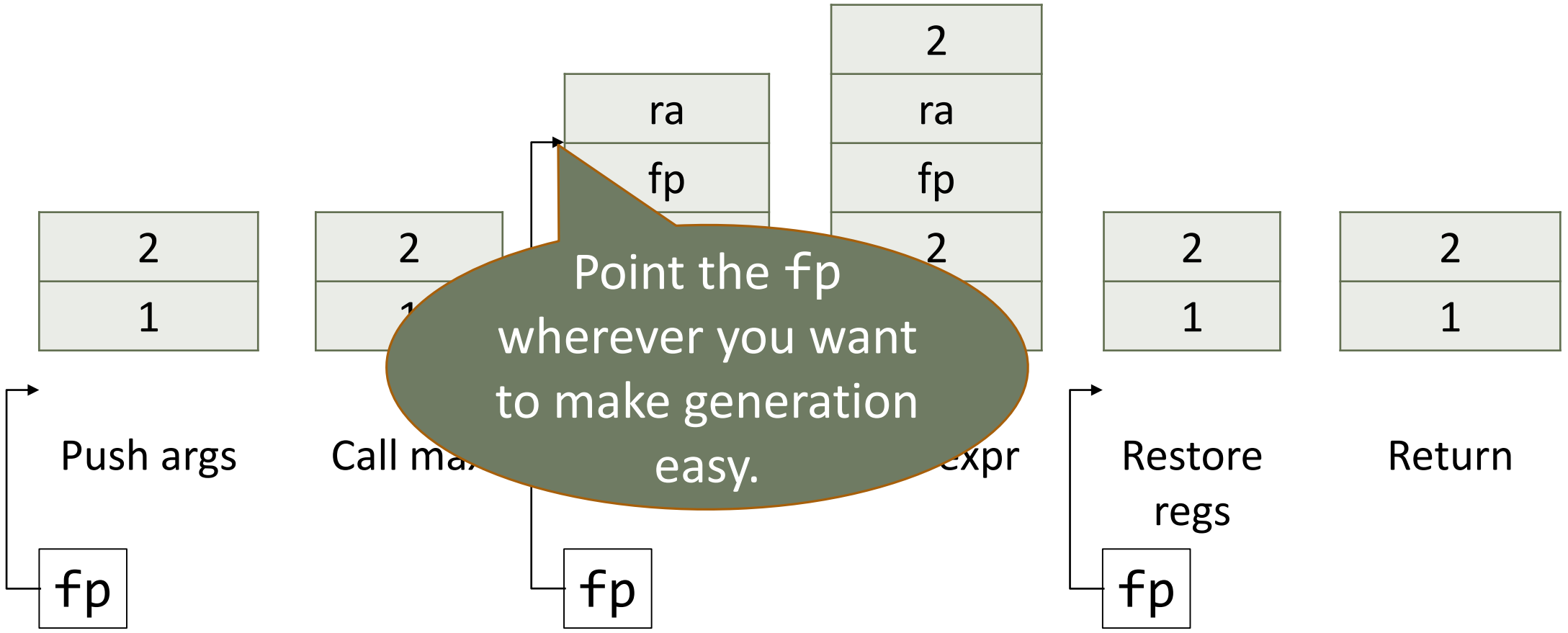
Function Call Example: Syntax Trees



Function Call Example: Stack Discipline



Function Call Example: Stack Discipline



Closing Thoughts

Simple functions (like `max`) do not need a stack frame.

- Avoids saving and restoring registers.
- Avoids updating and restoring `fp`.
- More complicated code generation.

For performance, update `sp` once at start.

- Access temporaries and locals via explicit offsets from `fp`.

Objects

```
expr ::= ID <- expr
       | expr[@TYPE].ID ( [ expr [[, expr]]* ] )
       | ID( [ expr [[, expr]]* ] )
       | if expr then expr else expr fi
       | while expr loop expr pool
       | { [[expr; ]]+ }
       | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]]* in expr
       | case expr of [[ID : TYPE => expr; ]]+ esac
       | new TYPE
       | isvoid expr
       | expr + expr
       | expr - expr
       | expr * expr
       | expr / expr
       | ~expr
       | expr < expr
       | expr <= expr
       | expr = expr
       | not expr
       | (expr)
       | ID
       | integer
       | string
       | true
       | false
```

Implementation Considerations

How to lay out object in memory?

How to implement inheritance?

How to implement dynamic dispatch?

Struct Layout

Lay out fields contiguously.

- Each field at fixed offset.

Insert padding where needed for ***alignment***.

Field	Offset
Attribute 1	0
Attribute 2	1
…	…
Attribute N	N

Alignment?

Data may only be read from some subset of addresses.

On x86_64 address must be multiple of data size.

- 8-byte pointers must have address divisible by 8.
- 4-byte ints must have address divisible by 4.
- Object tend to have alignment of largest field.

Not a concern for Cool.

Inheritance

Liskov Substitution Principle:

If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected.

The fields B inherits from A must have the same offsets.

Field	Offset
Attribute A.1	0
Attribute A.2	1
…	…
Attribute A.N	N
Attribute B.1	N+1
Attribute B.2	N+2
…	…

Static Dispatch

Like function calls with two modifications:

1. Pass “self” as implicit parameter.

2. Place fields in “self” object into symbol table.

Dynamic Dispatch

```
Class A {
   f(): Object {
      out_string(“A”)
   }
   g(): Object {
      f()
   }
}
```

```
Class B inherits A {
   f(): Object {
      out_string(“B”)
   }
}
```

Dynamic Dispatch

What does `e.g()` print?

- If `e` is an A: “A”
- If `e` is a B: “B”

`g()` must work for ***both***.

Need to look up method label in object at ***run time***, not compile time.

How?

More fields.

Implementing Dynamic Dispatch

Methods are same for all instances of class.

- Carrying copies of labels in all objects is redundant.

Instead use one ***virtual function table*** (vtable) per class instead.

Object Layout	Offset
vtable	0
Attribute 1	1
…	…

vtable Layout	Offset
Method 1	0
…	…

Dynamic Dispatch Example

Calling conventions:

- Self object pointer passed on stack before arguments.
- Arguments passed on stack. Arg 1 is below arg N.
- Return value in r1.
- All other registers are callee saved.

Dispatch Tables		
Offset	A	B
0	A.f	B.f
1	A.g	A.g

Dynamic Dispatch Example

```
A.g:
    push ra
    ld r1 <- sp[1] ; get self obj
    push r1        ; pass self arg
    ld r1 <- r1[0] ; get vtable
    ld r1 <- r1[0] ; get f() label
    call r1
    ; return value now in r1
```

```
  pop ra ; self obj
  pop ra ; return addr
  return
```

Multiple Inheritance