CS 6015: Software Engineering

Spring 2024

Lecture 17: Functions (Project Related)

This Week

- Functions (Project related)

Next Week

- Undefined behavior
- Smart/shared pointers

MSDscript: New extension

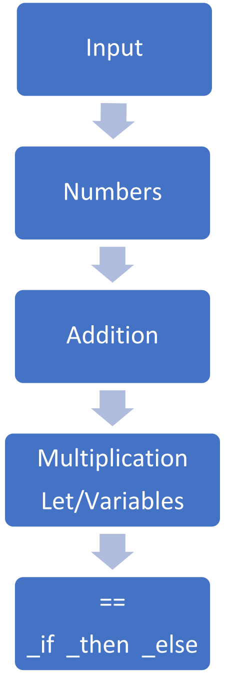
Grammar

```
⟨expr⟩  =  ⟨number⟩
           |  ⟨expr⟩ + ⟨expr⟩
           |  ⟨expr⟩ * ⟨expr⟩
           |  ⟨variable⟩
           |  _let  ⟨variable⟩ = ⟨expr⟩  _in  ⟨expr⟩
           |  _if  ⟨expr⟩  _then  ⟨expr⟩  _else  ⟨expr⟩

           |  Function  _fun (variable) <expr>
           |   Call function <expr> ( <expr> )
```

Add new functionality
for our MSDscript

Parsing



Update the parser to
parse the new grammar

Functions in Algebra

```
f(x) = x + 1

f(10)
```

Functions in JavaScript

```
≫ function f(x) { return x + 1; }
≫ f(10)
≫ f
≫ [1, 2, 3].map(f)
≫ var f = function (x) { return x + 1; }
≫ f(10)
≫ [1, 2, 3].map(f)
≫ [1, 2, 3].map(function (x) { return x + 1; })
≫ (function (x) { return x + 1; })(10)
```

From JavaScript to MSDscript

JavaScript:

```
f(10)
```

MSDscript:

```
f(10)
```

From JavaScript to MSDscript

JavaScript:

```
function (x) { return x + 1; }
```

MSDscript:

```
_fun (x) x + 1
```

Functions in MSDscript

```
_let f = _fun (x) x + 1
_in  f(10)
```

➡ `11`

```
⟨expr⟩  =  ....
        |  _fun ( ⟨variable⟩ ) ⟨expr⟩
        |  ⟨expr⟩ ( ⟨expr⟩ )
```

Functions in MSDscript

```
_let f = _fun (x) x + 1
_in  f(10)
```

➡ `11`

```
⟨expr⟩  =  ....
        |  _fun ( ⟨variable⟩ ) ⟨expr⟩
        |  ⟨expr⟩ ( ⟨expr⟩ )
```

Any ⟨expr⟩, not just ⟨variable⟩s

Functions in MSDscript

```
_let f = _fun (x) x + 1
_in  f(10)
```

➡ `11`

```
⟨expr⟩ = ....
       | _fun ( ⟨variable⟩ ) ⟨expr⟩      FunExpr
       | ⟨expr⟩ ( ⟨expr⟩ )               CallExpr
```

Functions in MSDscript

```
_let f = _fun (x) x + 1
_in  f(10)
```

➡ 11

```
⟨expr⟩ = ....
       | _fun ( ⟨variable⟩ ) ⟨expr⟩      FunExpr
       | ⟨expr⟩ ( ⟨expr⟩ )               CallExpr
```

```
class FunExpr : public Expr {
  std::string formal_arg;
  Expr *body;
}
```

```
class CallExpr : public Expr {
  Expr *to_be_called;
  Expr *actual_arg;
}
```

Functions in MSDscript

```
_fun (x) x + 1

➡ _fun (x) x + 1
```

```
⟨val⟩  =  ⟨number⟩                              NumVal
       |  ⟨boolean⟩                             BoolVal
       |  _fun ( ⟨variable⟩ ) ⟨expr⟩            FunVal
```

Functions in MSDscript

```
_fun (x) x + 1
```

➡ `_fun (x) x + 1`

⟨val⟩	=	⟨number⟩	**NumVal**
	|	⟨boolean⟩	**BoolVal**
	|	`_fun` `(` ⟨variable⟩ `)` ⟨expr⟩	**FunVal**

```
class FunVal : public Val {
  std::string formal_arg;
  Expr *body;
}
```

Functions in MSDscript

```
_fun (x) x + 1
```

```
➡ _fun (x) x + 1
```

```
⟨val⟩ = ⟨number⟩                          NumVal
      | ⟨boolean⟩                         BoolVal
      | _fun ( ⟨variable⟩ ) ⟨expr⟩        FunVal
```

```
class FunVal : public Val {
  std::string formal_arg;
  Expr *body;
}
```

This is new: an ***Expression*** inside a ***Value***

Functions in MSDscript

```
_fun (x) x + 1
```

➡ `_fun (x) x + 1`

Functions in MSDscript

```
_fun (x) x + 1
```

```
➡ _fun (x) x + 1
```

```
Val *FunExpr::interp() {
}
```

Functions in MSDscript

```
_fun (x) x + 1
```

```
➡ _fun (x) x + 1
```

```
Val *FunExpr::interp() {
  .... formal_arg ....
  .... body ...
}
```

Functions in MSDscript

```
_fun (x) x + 1
```

```
➡ _fun (x) x + 1
```

```
Val *FunExpr::interp() {
  .... formal_arg ....
  .... body->interp() ...
}
```

Functions in MSDscript

```
_fun (x) x + 1
```

```
➡ _fun (x) x + 1
```

```
Val *FunExpr::interp() {
  new FunVal(formal_arg,
             body);
}
```

Functions in MSDscript

```
_fun (x) x + 1
```

```
➡ _fun (x) x + 1
```

```
Val *FunExpr::interp() {
  return new FunVal(formal_arg,
                    body);
}
```

Functions in MSDscript

```
_let f = _fun (x) x + 1
_in  f(10)
```

Functions in MSDscript

```
(_fun (x) x + 1)(10)
```

➡ `11`

```
Val *CallExpr::interp() {
}
```

Functions in MSDscript

```
(_fun (x) x + 1)(10)
```

➡ `11`

```
Val *CallExpr::interp() {
  .... to_be_called ....
  .... actual_argument ....
}
```

Functions in MSDscript

```
(_fun (x) x + 1)(10)
```

➡ `11`

```
Val *CallExpr::interp() {
  .... to_be_called->interp() ....
  .... actual_argument->interp() ....
}
```

Functions in MSDscript

```
(_fun (x) x + 1)(10)
```

➡ `11`

```
Val *CallExpr::interp() {
  to_be_called->interp()
   ->call(actual_argument->interp());
}
```

```
class Val {
  virtual Val *call(Val *actual_arg) = 0;
}
```

Functions in MSDscript

```
(_fun (x) x + 1)(10)
```

➡ 11

```
Val *CallExpr::interp() {
  return
  to_be_called->interp()
   ->call(actual_argument->interp());
}
```

```
class Val {
  virtual Val *call(Val *actual_arg) = 0;
}
```

Functions in Algebra and MSDscript

```
f(x) = x*x

f(2)
```

```
_let f = _fun (x) x*x
_in  f(2)
```

Interpreting with Functions

```
_let f = _fun (x) x*x
_in  f(2)
```

➡ `(_fun (x) x*x)(2)`

➡ `2*2`

➡ `4`

Grammar with Functions and Calls

```
⟨expr⟩ = ⟨number⟩
       | ⟨boolean⟩
       | ⟨expr⟩ == ⟨expr⟩
       | ⟨expr⟩ + ⟨expr⟩
       | ⟨expr⟩ * ⟨expr⟩
       | ⟨expr⟩ ( ⟨expr⟩ )                      new
       | ⟨variable⟩
       | _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
       | _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
       | _fun ( ⟨variable⟩ ) ⟨expr⟩             new
```

Grammar with Functions and Calls

```
⟨expr⟩ = ⟨number⟩
       | ⟨boolean⟩
       | ⟨expr⟩ == ⟨expr⟩
       | ⟨expr⟩ + ⟨expr⟩
       | ⟨expr⟩ * ⟨expr⟩
       | ⟨expr⟩ ( ⟨expr⟩ )   new
       | ⟨variable⟩
       | _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
       | _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
       | _fun ( ⟨variable⟩ ) ⟨expr⟩   new
```

Higher precedence than `*`

`2 * f(3)` ≡ `2 * (f(3))`

Grammar with Functions and Calls

```
⟨expr⟩  =  ⟨number⟩
        |  ⟨boolean⟩
        |  ⟨expr⟩ == ⟨expr⟩
        |  ⟨expr⟩ + ⟨expr⟩
        |  ⟨expr⟩ * ⟨expr⟩
        |  ⟨expr⟩ ( ⟨expr⟩ )   new
        |  ⟨variable⟩
        |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
        |  _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
        |  _fun ( ⟨variable⟩ ) ⟨expr⟩   new
```

Left-associative

`f(3)(2)  ≡  (f(3))(2)`

Parsing with Conditions and Comparisons

```
⟨expr⟩       =  ⟨comparg⟩
             |  ⟨comparg⟩ == ⟨expr⟩

⟨comparg⟩    =  ⟨addend⟩
             |  ⟨addend⟩+ ⟨comparg⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
             |  _true
             |  _false
             |  _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
```

Parsing with Functions and Calls

```
⟨expr⟩       =  ⟨comparg⟩
             |  ⟨comparg⟩ == ⟨expr⟩

⟨comparg⟩    =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨comparg⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨inner⟩
             |  ⟨multicand⟩ ( ⟨expr⟩ )

⟨inner⟩      =  ⟨number⟩ | ( ⟨expr⟩ ) | ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
             |  _true | _false
             |  _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
             |  _fun ( ⟨variable⟩ ) ⟨expr⟩
```

Parsing with Functions and Calls

```
⟨expr⟩       = ⟨comparg⟩
             | ⟨comparg⟩ == ⟨expr⟩

⟨comparg⟩    = ⟨addend⟩
             | ⟨addend⟩ + ⟨comparg⟩

⟨addend⟩     = ⟨multicand⟩
             | ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  = ⟨inner⟩
             | ⟨multicand⟩ ( ⟨expr⟩ )

⟨inner⟩      = ⟨number⟩ | ( ⟨expr⟩ ) | ⟨variable⟩
             | _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
             | _true | _false
             | _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
             | _fun ( ⟨variable⟩ ) ⟨expr⟩
```

```
parse_multicand() {
  expr = parse_inner()
  while (in.peek() == '(') {
    consume(in, '(')
    actual_arg = parse_expr()
    consume(in, ')')
    expr = new CallExpr(expr,
                        actual_arg)
  }
  return expr
}
```

Functions and Other Variables

```
y = 8
f(x) = x*y

f(2)
```

```
_let y = 8
_in  _let f = _fun (x) x*y
     _in  f(2)
```

Interpreting with Variables and Functions

```
_let y = 8
_in  _let f = _fun (x) x*y
     _in  f(2)
```

Interpreting with Variables and Functions

```
_let y = 8
_in  _let f = _fun (x) x*y
     _in  f(2)

➡ _let f = _fun (x) x*8
  _in  f(2)
```

Interpreting with Variables and Functions

```
_let y = 8
_in  _let f = _fun (x) x*y
     _in  f(2)

➡ _let f = _fun (x) x*8
  _in  f(2)

➡ 2*8
```

Interpreting with Variables and Functions

```
_let y = 8
_in  _let f = _fun (x) x*y
     _in  f(2)

➡ _let f = _fun (x) x*8
  _in  f(2)

➡ 2*8

➡ 16
```

Interpreting with Variables and Functions

```
_let x = 8
_in  _let f = _fun (x) x*x
     _in  f(2)
```

Interpreting with Variables and Functions

```
_let x = 8
_in  _let f = _fun (x) x*x
     _in  f(2)

➡ _let f = _fun (x) x*x
  _in  f(2)
```

Interpreting with Variables and Functions

```
_let x = 8
_in  _let f = _fun (x) x*x
     _in  f(2)

➡ _let f = _fun (x) x*x
  _in  f(2)

➡ 2*2
```

Interpreting with Variables and Functions

```
_let x = 8
_in  _let f = _fun (x) x*x
     _in  f(2)

➡ _let f = _fun (x) x*x
  _in  f(2)

➡ 2*2

➡ 4
```

Local Binding vs. Functions

```
_let x = 1
_in  x+2
```

```
(_fun (x) x+2)(1)
```

Local Binding vs. Functions

```
_let x = 1
_in  x+2

➡ 1+2
```

```
(_fun (x) x+2)(1)

➡ 1+2
```

Local Binding vs. Functions

```
_let var = rhs
_in  body
```

```
(_fun (var) body) (rhs)
```

Local Binding vs. Functions

`_let` *var* **`=`** *rhs*
`_in` *body*

`(_fun` **`(`***var***`)`** *body***`)`** **`(`***rhs***`)`**

So, **`_let`** is technically unnnecessary
— but often more convenient

Multiple Arguments vs. Currying

```
f(x, y) = x*x + y*y

f(2, 3)
```

```
_let f = _fun (x)
          _fun (y)
            x*x + y*y
_in  f(2)(3)
```

Multiple Arguments vs. Currying

```
f(x, y) = x*x + y*y

f(2, 3)
```

```
_let f = (_fun (x)
           (_fun (y)
              x*x + y*y))
_in  (f(2))(3)
```

Interpreting Curried Functions

```
_let f = (_fun (x)
           (_fun (y)
              x*x + y*y))
_in  (f(2))(3)

➡  ((_fun (x)
       (_fun (y)
          x*x + y*y))(2))(3)

➡  (_fun (y)
      2*2 + y*y)(3)

➡  2*2 + 3*3

➡  13
```

Partial Application

```
_let add = _fun (x)
             _fun (y)
               x + y
_in  add(5)(10)
```

Partial Application

```
_let add = _fun (x)
             _fun (y)
               x + y
_in  _let addFive = add(5)
     _in  addFive(10)
```

Partial Application

```
_let add = _fun (x)
            _fun (y)
              x + y
_in   _let addFive = add(5)
      _in  addFive(10)

➡  _let addFive = (_fun (x)
                     _fun (y)
                       x + y)(5)
   _in  addFive(10)
```

Partial Application

```
_let add = _fun (x)
            _fun (y)
              x + y
_in  _let addFive = add(5)
     _in  addFive(10)

➡ _let addFive = (_fun (x)
                    _fun (y)
                      x + y)(5)
  _in  addFive(10)

➡ _let addFive = _fun (y)
                    5 + y
  _in  addFive(10)
```

Recursive Functions

```
_let factorial = _fun (x)
                  _if x == 1
                  _then 1
                  _else x * factorial(x + -1)
_in factorial(5)
```

Doesn't work

`factorial` is visible only after `_in`

`factorial` is visible here

Recursive Functions

```
_let factrl = _fun (factrl)
                _fun (x)
                  _if x == 1
                  _then 1
                  _else x * factrl(factrl)(x + -1)
_in factrl(factrl)(5)
```

Recursive Functions

```
_let factrl = _fun (factrl)
                _fun (x)
                  _if x == 1
                  _then 1
                  _else x * factrl(factrl)(x + -1)
_in _let factorial = _fun (x)
                       factrl(factrl)(x)
_in factorial(5)
```

Recursive Functions

```
_let factrl = _fun (factrl)
                _fun (x)
                  _if x == 1
                  _then 1
                  _else x * factrl(factrl)(x + -1)
_in _let factorial = factrl(factrl)
_in factorial(5)
```

Recursive functions: More examples

```
_let fib = _fun (fib)
             _fun (x)
               _if x == 0
               _then 1
               _else _if x == 1
                     _then 1
                     _else fib(fib)(x + -2) + fib(fib)(x + -1)
_in  fib(fib)(30)
```

A Substitution Bug

```
_let f = _fun (x) x+y
_in _let y = 10
_in f(1)
```

A Substitution Bug

```
_let f = _fun (x) x+y
_in _let y = 10
_in f(1)
```

Free y

A Substitution Bug

```
_let f = _fun (x) x+y
_in _let y = 10
_in f(1)
```

Free y

```
➡ _let y = 10
  _in (_fun (x) x+y)(1)
```

A Substitution Bug

```
_let f = _fun (x) x+y
_in _let y = 10
_in f(1)
```

Free y

Bound y

```
➡ _let y = 10
  _in (_fun (x) x+y)(1)
```

A Substitution Bug

```
_let f = _fun (x) x+y
_in _let y = 10
_in f(1)
```

Free **y**

Bound **y**

```
➡ _let y = 10
  _in (_fun (x) x+y)(1)

➡ (_fun (x) x+10)(1)
```

For now, don’t try to fix this bug.
Instead, just avoid free variables in examples.