Designing a Simple Functional Languages: LET and LETREC

Lecture 8-10

(EOPL: Chapter 3)

LET and LETREC Languages

- Purely functional languages
  - no side-effects
- Designed to
  - Illustrate binding and scoping of variables
  - Illustrate how to specify the operational semantics of a language
  - Illustrate how to write an interpret for a language
  - Illustrate how to perform static type checking
  - Illustrate how to generate low-level code

Sample programs in LETREC

```
letrec add5(x) = x + 5

letrec fac(x) =
        if x == 1 then 1 else x*(fac (x-1))
in
        (fac 4)

letrec apply(f) = (f 10) in letrec inc(x) = x + 1 in apply(inc)

letrec adder(x) = letrec f(y) = x + y in f in ((adder 3) 4)
```

LET: An expression language

- The LET language is an expression language
- Expressions can be
  - integers: 1, 2, 3, 0
  - variables: x, y, foo, name, s2, y3
  - arithmetic expressions: 1 + 2 * 3 - 4
  - boolean expression: 3 > 6*x
  - conditional expressions: if x > y then x else y
  - let-in expressions: let x = 3 in x + 1

Values

- Expressed values: Int + Bool
  - possible values of expressions
- Denoted values: Int + Bool
  - values bound to variables
- Example of Int values: -2, 0, 3
- Bool values: true and false
- Language allows to only express Int values
- No literals for Bool values

Arithmetic and Boolean expressions

- Arithmetic Expressions
  - Only +, -, /, * operators are supported
  - Usual precedence and associativity
  - Example: 2 + 3 * 5 - 1
- Boolean Expressions
  - Comparison between two arithmetic expressions using ==, !=, >, <, <=, >=
  - Example: 5 > 6 * 6

Conditional expressions

- if cond then expr1 else expr2
  - cond is first evaluated
  - if cond evaluates to true, then expr1 is evaluated
    - the value of the if-then-else expression is the value of expr1
  - otherwise, expr2 is evaluated
    - the value of the if-then-else expression if the value of expr2
  - Example: if x > 0 then x else 0-x

Variables and let … in … expressions

- let iden = expr in body
  - let and in are keywords
  - evaluates expr
  - binds the value of expression expr to the variable iden (shadowing any other binding)
  - body is evaluated with the above binding of iden
  - the value of the body gives the value of the entire let … in … expression
  - Example: let x = 3 in (x + 1)/2
  - Example: let x = 2 in let y = 3 in x + y + 1

Example program in LET

```
let
        x = 7
in
        let
                y = 2
        in
                let
                        y = let
                                x = x - 1
                            in
                                x-y
                in
                        x - 8 - y
```

How do we interpret a LET program?

- Need to evaluate expressions containing variables
  - Need to know the value associated with each variable
  - We do this by keeping those values in an *environment*
- An environment is a function
  - domain is a finite set of variables, and
  - range is the denoted values

Denoting an Environment

- We use some abbreviations when writing about environments
  - ρ ranges over environments.
  - [] denotes the empty environment.
  - [*var* = *val*]ρ denotes the environment extended with the mapping var = val
  - [*var1* = *val1*,*var2* = *val2*]ρ abbreviates [*var1* = *val1*]([*var2* = *val2*]ρ), etc.
  - [*var1* = *val1*, *var2* = *val2*, . . .] denotes the environment in which the value of *var1* is *val1*, etc.

Scoping

- Variable can appear in two contexts:
  - as references: x + 3
  - as declarations: let x = 4 in ...
- Declarations have limited scope
  - scoping rules determine the scope of a declaration
  - portion of the program where a declaration is valid is called the scope of the declaration
- Lexical scoping
  - scope can be determined statically
  - a reference to a variable refers to the closest declaration that surrounds the reference
  - inner declaration of a variable shadows the outer one
  - lexical scopes are nested: one lies entirely in another

Example program in LET

```
let
        x = 7
in
        let
                y = 2
        in
                let
                        y = let
                                x = x - 1
                            in
                                x-y
                in
                        x - 8 - y
```

Example program in LET

```
let                                                    Env: [ ]
        x = 7
in                                                     Env: [x = 7]
        let
                y = 2
        in                                             Env: [x = 7, y = 2]
                let
                        y = let
                                x = x - 1
                            in                         Env: [x=6, y=2]
                                x-y
                in                                     Env: [x=7, y = 4]
                        x - 8 - y
```

Formal definition of LET: grammar

```
prog: expr;

expr: num
    | iden
    | '(' expr ')'
    | expr ('*' | '/') expr
    | expr ('+' | '–') expr
    | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
    | 'let' iden '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr;
```

```
num: INT;
iden: ID;

ID : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..
'Z'|'0'..'9'|'_')* ;
INT : '0'..'9'+ ;
WS : (' '|' \r' | ' \t' | ' \n')+ -> skip ;
```

Operational Semantics

- We need a complete but not overly restrictive specification of the language
- There are many ways to specify programming language semantics
- They are all equivalent but some are more suitable to various tasks than others
- Operational semantics
  - Describes the evaluation of programs on an abstract machine
  - Most useful for specifying implementations
  - This is what we will use for our languages

Other Kinds of Semantics

- Denotational semantics
  - The meaning of a program is expressed as a mathematical object
  - Very elegant but quite complicated
- Axiomatic semantics
  - Useful for checking that programs satisfy certain correctness properties
    - e.g., that the quick sort function sorts an array
  - The foundation of many program verification systems

Introduction to Operational Semantics

- We introduce a formal notation to describe operational semantics
  - Using logical rules of inference
- Evaluation rules are of the form

$$Context \vdash e : v$$

(in the given Context, expression e evaluates to value v)

Example of Inference Rule for Operational Semantics

- Example:

$$\frac{\text{Context} \vdash e_1 : 5 \qquad \text{Context} \vdash e_2 : 7}{\text{Context} \vdash e_1 + e_2 : 12}$$

- In general the result of evaluating an expression depends on the result of evaluating its subexpressions
- The logical rules specify everything that is needed to evaluate an expression

What Contexts Are Needed?

- Obs.: Contexts are needed to handle variables
- Consider the evaluation of let x = 2 in (let x = 3 in x + 1) + 1
  - We need to keep track of values of variables
  - We need to allow variables to change their values during the evaluation
- We track variables and their values with:
  - An environment : tells us the value bound to a variable

Variable Environments

- A variable environment is a map from variable names to values
- Example:

$$E = [a = v_1, b = v_2]$$

- To lookup a variable a in environment E we write $E(a)$

LET Values

- Primitive values

Int(5)	the integer 5
Bool(true)	the boolean true

Operational Rules of LET

- Evaluation rules are of the form

$$\frac{\text{Hypothesis}_1 \ \ldots \ \text{Hypothesis}_n}{\text{Judgement}}$$

- The evaluation judgment is

$$E \vdash e : v$$

read:

- Given E the current variable environment
- If the evaluation of e terminates then
- The returned value is v

Notes

- The "result" of evaluating an expression is a value
- The variable environment does not change
- The operational semantics allows for non-terminating evaluations
- We define one rule for each kind of expression

Operational Semantics for Base Values and Arithmetic/Boolean Expressions

$$\frac{i \text{ is an integer literal}}{E \vdash i : Int(i)} \qquad \frac{E(id) = v}{E \vdash id : v}$$

$$\frac{E \vdash e1 : Int(v1) \quad E \vdash e2 : Int(v2) \quad v1 + v2 = v}{E \vdash e1 + e2 : Int(v)} \qquad \frac{E \vdash e1 : Int(v1) \quad E \vdash e2 : Int(v2) \quad (v1 > v2) = v}{E \vdash e1 > e2 : Bool(v)}$$

Operational Semantics of Conditionals

$$\frac{E \vdash e_1 : \text{Bool(true)} \quad E \vdash e_2 : v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v}$$

$$\frac{E \vdash e_1 : \text{Bool(false)} \quad E \vdash e_3 : v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v}$$

- The result of evaluating e_1 is a boolean value
  - Dynamic type checking must ensure this

Operational Semantics of let Expression

$$\frac{E \vdash e_1 : v_1 \quad ??\vdash e_2 : v_2}{E \vdash \text{let id} = e_1 \text{ in } e_2 : v_2}$$

- What is the context in which e_2 must be evaluated?

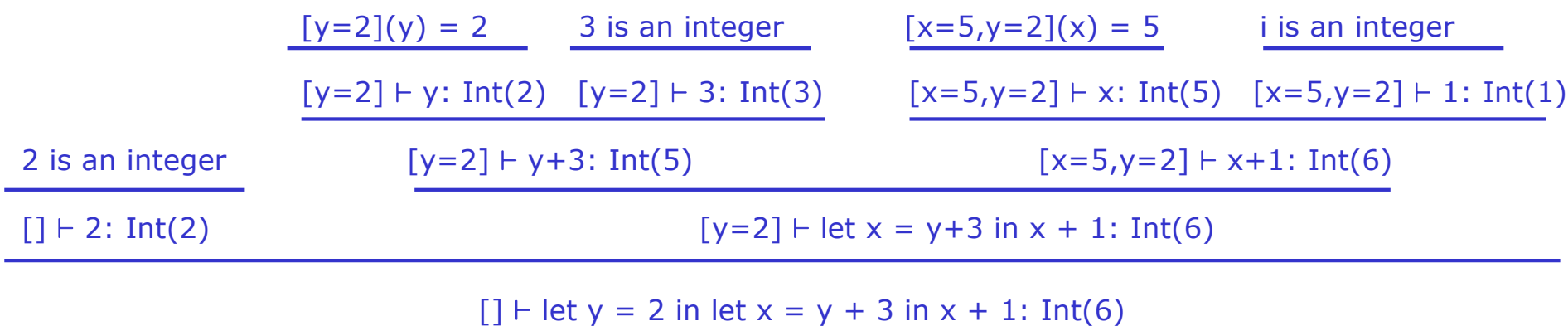
Operational Semantics of let Expression

$$\frac{E \vdash e_1 : v_1 \qquad [id = v_1]E \vdash e_2 : v_2}{E \vdash \text{let } id = e_1 \text{ in } e_2 : v_2}$$

- What is the context in which e_2 must be evaluated?
  - Environment like E but with a new binding of id to a to the value of e_1

Evaluation of an Expression: Inverted Tree

Expression evaluation using operational semantics rules can be expressed as an inverted tree:



- The root of the tree is the whole expression
- Each node is an instance of an operational semantics rule
- Leaves are the rules with no hypotheses

Operational Semantics to Interpreter: Invert the Rules

- Define an eval function
  - eval takes as argument an expression and a context
  - returns the value of the expression

- if eval(e, E) = v, then $E \vdash e : v$

Operational Semantics to Interpreter: Invert the Rules

$$\frac{E(id) = v}{E \vdash id : v}$$

```
eval(id, E):
        return E(id)
```

$$\frac{E \vdash e1 : Int(v1) \quad E \vdash e2 : Int(v2) \quad v1 + v2 = v}{E \vdash e1 + e2 : Int(v)}$$

$$\frac{E \vdash e_1 : v_1 \quad [id = v_1]E \vdash e_2 : v_2}{E \vdash \text{let } id = e_1 \text{ in } e_2 : v_2}$$

Operational Semantics to Interpreter: Invert the Rules

$$\frac{E(id) = v}{E \vdash id : v}$$

```
eval(id, E):
        return E(id)
```

$$\frac{E \vdash e1 : Int(v1) \quad E \vdash e2 : Int(v2) \quad v1 + v2 = v}{E \vdash e1 + e2 : Int(v)}$$

```
eval(e1 + e2, E):
        v1 = eval(e1, E)
        v2 = eval(e2,E)
        return v1 + v2
```

$$\frac{E \vdash e_1 : v_1 \quad [id = v_1]E \vdash e_2 : v_2}{E \vdash \text{let } id = e_1 \text{ in } e_2 : v_2}$$

```
eval (let id = e1 in e2, E):
        v1 = eval(e1, E)
        v2 = eval(e2, extend(E,id,v1))
        return v2
```

Operational Semantics to Interpreter: Invert the Rules

$$\frac{E \vdash e_1 : \text{Bool(true)} \qquad E \vdash e_2 : v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v}$$

$$\frac{E \vdash e_1 : \text{Bool(false)} \qquad E \vdash e_3 : v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v}$$

Operational Semantics to Interpreter: Invert the Rules

$$\frac{E \vdash e_1 : \text{Bool(true)} \quad E \vdash e_2 : v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v}$$

$$\frac{E \vdash e_1 : \text{Bool(false)} \quad E \vdash e_3 : v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v}$$

```
eval(if e1 then e2 else e3, E):
        v1 = eval(e1, E)
        if (v1)
                v = eval(e2, E)
        else
                v = eval(e3, E)
        return v
```

LETREC: An expression language with lambdas

- The LETREC language is an expression language
- Expressions can be
  - integers: 1, 2, 3, 0
  - variables: x, y, foo, name, s2, y3
  - arithmetic expressions: 1 + 2 * 3 - 4
  - boolean expression: 3 > 6*x
  - conditional expressions: if x > y then x else y
  - let-in expressions: let x = 3 in x + 1
  - lambda expressions: letrec inc (x) = x + 1 in (inc 5)
    - lambdas are functions with no side-effects

Values for LETREC

- Expressed values: Int + Bool + Lambda
  - possible values of expressions
- Denoted values: Int + Bool + Lambda
  - values bound to variables
- Example of Int values: -2, 0, 3
- Bool values: true and false
- Lambda values: letrec inc(x) = x + 1 in (inc 3)
  - defines a lambda value and binds it to inc

letrec … in … expression and lambda application

- letrec iden1(iden2) = body in expr
- (expr 1 expr2)
- Examples

```
letrec add5(x) = x + 5 in (add5 6)

letrec fac(x) =
        if x == 1 then 1 else x*(fac (x-1))
in
        (fac 4)

letrec apply(f) = (f 10) in letrec inc(x) = x + 1 in (apply inc)

letrec adder(x) = letrec f(y) = x + y in f in ((adder 3) 4)
```

Variables and letrec … in … expressions

- letrec iden1(iden2) = body in expr
  - letrec and in are keywords
  - iden1(iden2) = body is a declaration: binds the value of the lambda to the variable iden1
  - expr is evaluated with the above binding of iden1
  - (e1 e2): if e1 evaluates to a lambda, then (e1 e2) applies the lambda to the value of e
    - first evaluates e2
    - then evaluates body with iden1 bound to the lambda and iden2 bound to the value of e2
    - the value of the body is the value of (e1 e2)

letrec example

- even(x) = if x == 0 then 1 else (odd (x-1))
- odd(x) = if x == 0 then 0 else (even (x-1))
- Mutually recursive definition: how can we express this in LETREC

letrec example

- even(x) = if x == 0 then 1 else (odd (x-1))
- odd(x) = if x == 0 then 0 else (even (x-1))
- Mutually recursive definition: how can we express this in LETREC

```
letrec
        even(x) =
           if x == 0
           then 1
           else letrec
                                odd(y) = if y == 0 then 0 else (even (y-1))
                 in
                                (odd (x - 1 ))
in
        (even 12)
```

Closures

- Declarations create bindings
  - let id = e1 in e2
  - letrec id1(id2) = e1 in e2
- Extent of binding without letrec can be determined statically: e.g. id is restricted to e2
- Extent of binding created by letrec are semi-infinite
  - letrec creates a closure that can be accessed outside e2: e.g. binding of id2 must be maintained outside e2
  - closure stores the lambda along with its environment
  - let f = letrec x = 3 in letrec add3(y) = y + x in add3 in (f 4)

Formal definition of LETREC: grammar

```
prog: expr;

expr: num
    | iden
    | '(' expr ')'
    | expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | expr ('==' | '!=' | '>' | '<' | '>=' | '<=') expr
    | 'let' iden '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr
    | 'letrec' iden '(' iden ')' = expr in expr
    | '(' expr expr ')'
```

```
num: INT;
iden: ID;

ID : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..
'Z'|'0'..'9'|'_')* ;
INT : '0'..'9'+ ;
WS : (' '|' \r' | ' \t' | ' \n')+ -> skip ;
```

LETREC Values

- Primitive values

  Int(5) the integer 5

  Bool(true) the boolean true

  Lambda(boundedVar, body, Environment)

Operational Semantics of letrec Expression

$$\frac{E' = [id_1 = \text{Lambda}(id_2, e_1, E')]\ E \qquad E' \vdash e_2 : v_2}{E \vdash \text{letrec } id_1(id_2) = e_1 \text{ in } e_2 : v_2}$$

$$\frac{E \vdash e_1 : \text{Lambda}(id, e', E') \qquad E \vdash e_2 : v \qquad [id = v]E' \vdash e' : v'}{E \vdash (e_1\ e_2) : v'}$$

- Note that e_2 is evaluated in E' where id_1 is bound to the lambda

Operational Semantics to Interpreter: Invert the Rules

$$\frac{E' = [id_1 = \text{Lambda}(id_2, e_1, E')]\ E \qquad E' \vdash e_2 : v_2}{E \vdash \text{letrec } id_1(id_2) = e_1 \text{ in } e_2 : v_2}$$

```
eval(letrec id1(id2) = e1 in e2, E):
        v = Lambda(id2, e1, _)
        E' = v.env = extend(E, id1, v)
        v2 = eval(e2, E')
        return v2
```

$$\frac{E \vdash e_1 : \text{Lambda}(id, e', E') \qquad E \vdash e_2 : v \qquad [id = v]E' \vdash e' : v'}{E \vdash (e_1\ e_2) : v'}$$

```
eval((e1 e2), E):
        let Lambda(id,e', E') = eval(e1, E)
        v = eval(e2, E)
        v' = eval(e', extend(E', id, v))
        return v'
```

Operational Semantics of letrec Expression

$$\frac{E' = [id_1 = \text{Lambda}(id_2, e_1, E)]\ E \qquad E' \vdash e_2 : v_2}{E \vdash \text{letrec } id_1(id_2) = e_1 \text{ in } e_2 : v_2}$$

$$\frac{E \vdash e_1 : \text{Lambda}(id, e', E') \qquad E \vdash e_2 : v \qquad [id = v]E' \vdash e' : v'}{E \vdash (e_1\ e_2) : v'}$$

- What happens if the environment in the Lambda is E instead of E'?

Runtime Errors

Operational rules do not cover all cases

Consider for example the following rules:

$$\frac{E \vdash e1 : Int(v1) \quad E \vdash e2 : Int(v2) \quad (v1 > v2) = v}{E \vdash e1 > e2 : Bool(v)}$$

What happens if *e1* evaluates to Bool or Lambda?

We will see that static type checking can catch such runtime errors before interpretation

Runtime Errors

Operational rules do not cover all cases

Consider for example the following rules:

$$\frac{E \vdash e_1 : \text{Lambda}(id, e', E') \quad E \vdash e_2 : v \quad [id = v]E' \vdash e' : v'}{E \vdash (e_1\ e_2) : v'}$$

What happens if *e1* evaluates to Bool or Int?

We will see that static type checking can catch such runtime errors before interpretation

Summary

- Operational rules are very precise
  - Nothing that matters is left unspecified
- Operational rules contain a lot of details
  - But not too many details
  - Read them carefully
- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential
  - But not always using the notation we used for LETREC