CS383 Tutorial #1

Course Project Introduction

Xusheng Luo

9/13/2016

Before we start

How many of you have used

- Java
- Eclipse/IntelliJ IDEA
- A functional language, e.g. Lisp / Haskell
  - Imperative vs. Functional?

Introduction

- **SimPL** (pronounced "*simple*")
  - simplified dialect of ML
  - both functional and imperative
- Write an *Interpreter* for **simPL**
  - What is an "interpreter"?

Introduction

- What is a **simPL** program like?

```
let
        add = fn x => fn y => x + y
in
        add 1 2
end
(* this is a comment *)
```

Task

Provide in skeleton:

1. Lexical Definition
2. Syntactic analysis

Write your own:

1. Type Inference
2. Evaluation

Lexical Definition

- The lexical definition of **SimPL** consists of four aspects:
  1. Comments
  2. Atoms
  3. Keywords
  4. Operators

```
let
        add = fn x => fn y => x + y
in
        add 1 2
end
(* this is a comment *)
```

Lexical Definition

- Comments
  - Comments in SimPL are enclosed by pairs of (* and *).
  - Comments are nestable, e.g. (* (* *) *) is a valid comment.
  - A comment can spread over multiple lines.
  - Comments and whitespaces (spaces, tabs, newlines) should be ignored and not evaluated.

Lexical Definition

- Atoms
  - Atoms are either integer literals or identifiers.
  - Integer literals are matched by regular expression [0-9]+.
  - Integer literals only represent non-negative integers less than 2^{31}.
  - Integer literals are in decimal format, and leading zeros are insignificant e.g. 0123 represent the integer 123.
  - Identifiers are matched by regular expression [_a-z][_a-zA-Z0-9']*.

Lexical Definition

- Keywords
- All the following identifiers are keywords.
  - ref
  - fn rec
  - let in end
  - if then else
  - while do
  - true false
  - not andalso orelse

Lexical Definition

- Operations
- All the following identifiers are operations.
  - + - * / % ~
  - = <> < <= > >=
  - :: () =>
  - := !
  - , ; ( )

Syntactic Analysis

- All SimPL programs are expressions. **Exp** is the set of all expressions.
- Names are non-keyword identifiers. **Var** is the set of all names.
- Expressions, names, and integer literals are denoted by meta variables *e*, *x*, and *n*.
- Unary operator *uop*∈{~, not, !}
- Binary operator *bop*∈{+, -, *, /, %, =, <>, <, <=, >, >=, andalso, orelse, ::, :=, ;}

Syntactic Analysis

Expression *e*	::=	*n*	integer literal
	|	*x*	name
	|	`true`	true value
	|	`false`	false value
	|	`nil`	empty list
	|	`ref` *e*	reference creation
	|	`fn` *x* `=>` *e*	function
	|	`rec` *x* `=>` *e*	recursion
	|	`(`*e*`,` *e*`)`	pair construction
	|	*uop e*	unary operation
	|	*e bop e*	binary operation
	|	*e e*	application
	|	`let` *x* `=` *e* `in` *e* `end`	binding
	|	`if` *e* `then` *e* `else` *e*	conditional
	|	`while` *e* `do` *e*	loop
	|	`()`	unit
	|	`(`*e*`)`	grouping

Syntactic analysis

- Operator Precedence

Priority	Operator(s)	Associativity
1	;	Left
2	:=	None
3	orelse	Right
4	andalso	Right
5	= <> < <= > >=	None
6	: :	Right
7	+ -	Left
8	* / %	Left
9	(application)	Left
10	~ not !	Right

Typing and Semantic

Arbitrary type t	::=	`int`
	|	`bool`
	|	`unit`
	|	t `list`
	|	t `ref`
	|	$t \times t$
	|	$t \rightarrow t$

Typing and Semantic

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{ref}\ e : t\ \texttt{ref}} \quad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash e_1 : t\ \texttt{ref} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : \texttt{unit}} \quad \text{(T-Assign)}$$

$$\frac{\Gamma \vdash e : t\ \texttt{ref}}{\Gamma \vdash !e : t} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \quad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t\ \texttt{list}}{\Gamma \vdash e_1 :: e_2 : t\ \texttt{list}} \quad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash e_1 : t_2 \to t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t_1} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x : t_1] \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : t_2} \quad \text{(T-Let)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad bop \in \{+, -, *, /, \%\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{int}} \quad \text{(T-Arith)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{bool}} \quad \text{(T-Rel)}$$

$$\frac{\Gamma \vdash e_1 : \alpha \quad \Gamma \vdash e_2 : \alpha \quad bop \in \{=, <>\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{bool}} \quad \text{(T-Eq)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \texttt{andalso}\ e_2 : \texttt{bool}} \quad \text{(T-AndAlso)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \texttt{orelse}\ e_2 : \texttt{bool}} \quad \text{(T-OrElse)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t} \quad \text{(T-Cond)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : \texttt{unit}} \quad \text{(T-Loop)}$$

Typing and Semantic

- Refer to the **project specification.**
- Get more **details**.

http://adapt.seiee.sjtu.edu.cn/~xusheng/cs383

Implementation

- Code in Java
- Submit a runnable JAR file, e.g. SimPL.jar
- Your interpreter is started by using
  - java -jar SimPL.jar *program*.spl
- Output the result of the execution to the standard output (System.out)
- Case time limit: 5000ms

Output

- Syntax error => syntax error
- Type error => type error
- Runtime error => runtime error
- Integer => its value
- tt => true
- ff => false
- Function => fun
- … (refer to the spec. file)

How to run

/src/simpl/interpreter/Interpreter.java:

```
void run(String filename) {
    try (InputStream inp = new FileInputStream(filename)) {
        Parser parser = new Parser(inp);
        java_cup.runtime.Symbol parseTree = parser.parse();
        Expr program = (Expr) parseTree.value;
        program.typecheck(new DefaultTypeEnv());
        System.out.println(program.eval(new InitialState()));
    }
    catch (SyntaxError e) {
        System.out.println("syntax error");
    }
    catch (TypeError e) {
        System.out.println("type error");
    }
    catch (RuntimeError e) {
        System.out.println("runtime error");
    }
}
```

Examples

```
let
        addFive = fn x => x + 5
in
        addFive 2
end
```

$f(x) = x+5$

$\lambda x.x+5$

Functions are first-class values

Examples

```
let
        add = fn x => fn y => x + y
in
        add 1 2
end
```

Only unary functions

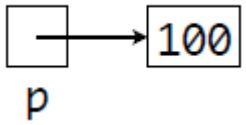
Examples

```
let
        factorial = rec f =>
                fn x => if x=1
                        then 1
                        else x * (f (x - 1))
in
        factorial 4
end
(* this is a comment *)
```

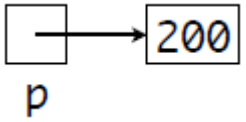
Examples

```
let
        p = ref 100
        (* p = new int(100); *)
in
        p := 200;
        (* *p = 200; *)
        !p
        (* return *p; *)
end
```

reference creation



assignment



dereference

Bonus

- Garbage collection
- Polymorphic type
- Lazy evaluation
- …

Tips

- Prepare early
- Do not panic
- Do it independently
- Try to earn bonus
- Be careful about the submit format

Conclusion

- Fully understand the specification before you write the code
- Resources are available online at

  http://www.cs.sjtu.edu.cn/~kzhu/cs383

  http://adapt.seiee.sjtu.edu.cn/~xusheng/cs383
- Send feedback to freefish_6174@126.com