Handout 1

Interpretation and Compilation
23-OCT-2018
due
9-NOV-2018: 23:59

Luis Caires

Goal

Implement a complete interpreter for the basic imperative-functional language specified

Use the approach developed in the lectures

- LL(1) parser using JAVACC
- AST model
- Environment based evaluator
- Dynamic type checking – issue proper error messages for runtime type errors

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

Submission Instructions

Create a bitbucket repository

Add me (lcaires@fct.unl.pt) as a team member

Send me the repository URL in an email with subject

ICL HO1 XXXXX YYYYY

where XXXXX etc are the student numbers (members of the group)

Abstract Syntax

```
EE -> EE ; EE | EE := EE
    | num | id | bool | let (id = EE)+ in EE end
    | fun id*-> EE end
    | EE ( EE* )
    | new EE | <!> EE
    | if EE then EE else EE end
    | while EE do EE end
    | EE binop EE
    | unop EE
```

Concrete Syntax

```
EM -> E(<;>EM)*                              ASTSeq(E1,E2)
E -> EA(< == > EA)?                          ASTEq(EA,EA)
EA -> T(<+>EA)*                              ASTAdd(E1,E2)
T -> F ( (<*>T)*                             ASTMul(F,T)
       | (<(>AL<)>)*                         ASTApply(F,AL)
       | <:=> E)                             ASTAssign(F,E)
AL -> (EM(<,>EM)*)?
PL -> (id(<,>id)*)?
F -> num | id | bool | let (id = EM)+ in EM end
  | fun PL -> EM end | <(> EM <)>
  | new F | <!> F
  | if EM then EM else EM end                ASTIf(EM,EM,EM)
  | while EM do EM end                       ASTWhile(EM,EM)
```

Basic operations

Arithmetic operations (on integer values)

E+E, E-E, E*E, E/E, -E

Relational operations

E==E, E>E, E<E, E<=E, E>=E

Logical operations (on boolean values)

E && E, E || E, ~E

AST(schematic)

```
interface ASTNode {
IValue eval(Environment env) ...
}



class AST??? implements ASTNode {


}
```

IValues (schematic)

```
interface IValue {
void show();
}

//Value constructors
VInt(n)
Closure(args,body,env)
VBool(t)
VCell(value)
```

IValues (schematic)

```
class VInt implements IValue {
int v;
VInt(int v0) { v = v0; }
int getval() { return v;}
}
```

IValues (schematic)

```
class VCell implements IValue {
IValue v;
VCell(IValue v0) { v = v0; }
IValue get() { return v;}
void set(IValue v0) { v = v0;}
}
```

IValues (schematic)

```
class ASTAdd implements ASTNode {

IValue eval(Environmnent env) {
v1 = left.eval(env);
if (v1 instanceof  VInt) {
   v2 = right.eval(env)
   if (v2 instanceof VInt) {
        return new Vint((VInt)v1).getval()+((VInt)v2).getval())
}
throw TypeError("illegal arguments to + operator");
}
```

Examples

```
(new 3) := 6;;

let a = new 5 in a := !a + 1; !a end;;

let x = new 10
    s = new 0  in
while !x>0 do
    s := !s + !x ; x := !x – 1
end; !s
end;;
```

Examples

```
let  f = fun n, b->
          let
            x = new n
            s = new b
          in
              while !x>0 do
                s := !s + !x ; x := !x – 1
              end;
              !s
          end
        end
in f(10,0)+f(100,20)
end;;
```

Typed Language

Interpretation and Compilation
15-NOV-2018

Luis Caires

Concrete Syntax (Typed Language)

Ty ->	**int**	ASTIntType()
|	**bool**	ASTIntType()
|	**ref** Ty	ASTIntType(Ty)
|	(Ty,...,Ty)Ty	ASTFunType(List<Ty>,Ty)

Concrete Syntax (Typed Language)

```
EM -> E(<;>EM)*                                  ASTSeq(E1,E2)
E -> EA(< == > EA)?                              ASTEq(EA,EA)
EA -> T(<+>EA)*                                  ASTAdd(E1,E2)
T -> F ( (<*>T)*                                 ASTMul(F,T)
       | (<(>AL<)>)*                             ASTApply(F,AL)
       | <:=> E)                                 ASTAssign(F,E)
AL -> (EM(<,>EM)*)?
PL -> (id:Type(<,>id:Type)*)?
F -> num | id | bool | let (id : Type = EM)+ in EM end
  | fun PL -> EM end | <(> EM <)>
  | new F | <!> F
  | if EM then EM else EM end                    ASTIf(EM,EM,EM)
  | while EM do EM end                           ASTWhile(EM,EM)
```

Goal

Implement a complete type checker for the basic imperative-functional language specified

Use the approach developed in the lectures

- extend parser to support type declarations
- AST model for types
- Environment based typechecker
- Integrate with your interpreter, before running the program, typecheck it!

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

Examples

```
(new 3) := 6;;

let a : ref int = new 5 in a := !a + 1; !a end;;

let x : ref int = new 10
    s :ref  int = new 0  in
while !x>0 do
    s := !s + !x ; x := !x – 1
end; !s
end;;
```

Examples

```
let  f :( int,int)int = fun n:int, b:int->
           let
             x : ref int = new n
             s : ref int = new b
           in
              while !x>0 do
                s := !s + !x ; x := !x – 1
              end;
              !s
           end
         end
in f(10,0)+f(100,20)
end;;
```

Final Handout
Compiler

Interpretation and Compilation
3-DEC-2018

Luis Caires

Goal

Implement a compiler for the basic imperative-functional language specified

Use the approach developed in the lectures

- Define a compile method in interface ASTNode to transverse the AST and generate code
- Use type information (from the typechecker) as needed to generate proper code
- code generation for the JVM (assemble with Jasmin)

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

Levels of Acomplishment

Implement a compiler for the basic imperative-functional language specified

1 – Cover just the basic imperative language

2 – Cover the language with functions

3 – Cover the extension

The 3 languages are described in the next slides

Level 1

```
EM -> E(<;>EM)*                        ASTSeq(E1,E2)
E -> EA(< == > EA)?                    ASTEq(EA,EA)
EA -> T(<+>EA)*                        ASTAdd(E1,E2)
T -> F ( (<*>T)*                       ASTMul(F,T)
     | <:=> E)                         ASTAssign(F,E)
AL -> (EM(<,>EM)*)?
PL -> (id:Type(<,>id:Type)*)?
F -> num | id | bool | let (id : Type = EM)+ in EM end
   | new F | <!> F
   | <(> EM <)>
   | if EM then EM else EM end         ASTIf(EM,EM,EM)
   | while EM do EM end                ASTWhile(EM,EM)
   | println E                         ASTPrint(E)
```

Level 2

```
EM -> E(<;>EM)*                              ASTSeq(E1,E2)
E -> EA(< == > EA)?                          ASTEq(EA,EA)
EA -> T(<+>EA)*                              ASTAdd(E1,E2)
T -> F ( (<*>T)*                             ASTMul(F,T)
       | (<(>AL<)>)*                         ASTApply(F,AL)
       | <:=> E)                             ASTAssign(F,E)
AL -> (EM(<,>EM)*)?
PL -> (id:Type(<,>id:Type)*)?
F -> num | id | bool | let (id : Type = EM)+ in EM end
   | new F | <!> F
   | fun PL -> EM end | <(> EM <)>
   | if EM then EM else EM end            ASTIf(EM,EM,EM)
   | while EM do EM end                   ASTWhile(EM,EM)
   | println E                            ASTPrint(E)
```

Level 3

Level 3 language introduces a data type of records and a data type of strings

The syntax for record expressions is

```
[ id = E; id = E; id = E ]    // record construction
R.id                           // record label selection
```

Level 3 - Example

Example

```
let
   person1 = [ name = “joe”; age = 22 ]
   person2 = [ name = “mary”; age = 5]
in
  println  person1.age + person2.age
end
```

NOTE: this program prints out the value 27

Levels of Acomplishment

1 – Cover just the basic imperative language

worth 16/20 points in final handout grading

2 – Cover the language with functions

worth 18/20 points in final handout grading

3 – Cover the extension

worth 20/20 points in final handout grading

Due date for final handout:

17 December 2018