Error handling - Semantics - Branches

Compilers: Principles And Practice

Tiark Rompf

Where Were We?

What did we learn in the last class?

Operator Precedence

Can someone remind us of the algorithm?

Scala Highlight

```scala
abstract class Exp
case class Lit(x: Int) extend Exp
```

Scala Highlight

```
abstract class Exp
case class Lit(x: Int) extend Exp

// Java
abstract class Exp
class Lit extends Exp {
  private int x;
  public Lit(int x) {
    this.x = x
  }
  static Exp apply(int x) {
    return new Lit(x)
  }
}
```

Scala Highlight

```scala
val lit = Lit(1) // shortcut for Lit.apply(1)
val lit2 = Lit(1)

lit == lit2  // true
```

Scala Highlight

```
val lit = Lit(1) // shortcut for Lit.apply(1)
val lit2 = Lit(1)

lit == lit2  // true

// Java
Lit lit = Lit.apply(1);
Lit lit2 = Lit.apply(1);

lit == lit2 // false

// Need to add and use:
boolean equals(obj: Object) {
  return (obj instanceof Lit) && ((Lit) obj).x == x;
}
```

Current Grammar (extended from last lecture)

```
<num>   ::= [0-9]+
<ident> ::= (a-zA-Z)[a-zA-Z0-9]*
<op>    ::= ['+' | '-' | '*' | '/']+
<atom>  ::= <num>
          | <ident>
          | '('<simp>')'
<simp>  ::= <atom>[<op><atom>]*
<exp>   ::= <simp>
          |'val' <ident> '=' <simp>';' <exp>
```

Quiz

What kind of error(s)?

```
val x = z; val 1 = 2; x
```

Quiz

What kind of error(s)?

```
val x = z; val 1 = 2; x
```

```
val x = 1++3; val y = x & 1
```

Quiz

What kind of error(s)?

```
val x = z; val 1 = 2; x
```

```
val x = 1++3; val y = x & 1
```

```
val x = 1; val x = 3; x + x
```

Quiz

What kind of error(s)?

```
val x = z; val 1 = 2; x

val x = 1++3; val y = x & 1

val x = 1; val x = 3; x + x

Answer:

 - Syntax error: identifier expected got 1.
   Semantic error: undefined identifier 'z'
 - Syntax error: unexpected character '&'.
   Semantic error: undefined operator '++'
 - We need more information to conclude. No errors based on previous
   lecture's interpreter. We could choose to forbid redefining variables.
```

Error Handling

- The parser handles the syntax errors.

```
Error: identifier expected.
1:16: val x = z; val 1 = 2; x
                     ^
```

Error Handling

- The parser handles the syntax errors.

```
Error: identifier expected.
1:16: val x = z; val 1 = 2; x
                     ^
```

- The semantic analyzer handles the semantic errors.

```
Error: undefined reference to z.
1:9: val x = z; val y = 2; x
             ^
1 error found
```

Error Handling

- The parser handles the syntax errors.

```
Error: identifier expected.
1:16: val x = z; val 1 = 2; x
                     ^
```

- The semantic analyzer handles the semantic errors.

```
Error: undefined reference to z.
1:9: val x = z; val y = 2; x
             ^
1 error found
```

- After these two phases, any problem is a compiler bug!

Token Position Information

```scala
// Position in the source code
case class Position(lineStart: Int, lineStop: Int,
                    colStart: Int, colEnd: Int)
abstract class Token {
  var pos: Position = _ // uninitialized
}
```

When the **Scanner** creates new tokens, it needs to assign the position of the token.

- Keep track of the number of lines read so far
- Keep track of the beginning of the lines (for column count)

Syntax Error Handling

- What to do when we find a syntax error?

Syntax Error Handling

- What to do when we find a syntax error?

Different solutions:

- Report the error and exit

Syntax Error Handling

- What to do when we find a syntax error?

Different solutions:

- Report the error and exit
- Try to recover and continue

Syntax Error Handling

- What to do when we find a syntax error?

Different solutions:

- Report the error and exit
- Try to recover and continue

Syntax Error Handling

- What to do when we find a syntax error?

Different solutions:

- Report the error and exit
- Try to recover and continue

For the project we are going to use the first method. But there are some algorithms which exist for error repair.

Syntax Error Repair

```
val 1 = 2; 4
```

Syntax Error Repair

```
val 1 = 2; 4
```

After ‘val’ we expect an identifier and find a number. We can raise an error, then create a ‘dummy’ identifier and continue parsing.

Syntax Error Repair

```
val 1 = 2; 4
```

After ‘val’ we expect an identifier and find a number. We can raise an error, then create a ‘dummy’ identifier and continue parsing.

Other algorithms exist: Burke-Fisher error repair is one of them.

Semantic Error Handling

```
val x = z; ++z
```

Semantic Error Handling

```
val x = z; ++z
```

```
Let("x", Ref("z"), Unary("++", Ref("z")))
```

Semantic Error Handling

```
val x = z; ++z
```

```
Let("x", Ref("z"), Unary("++", Ref("z")))
```

- 3 errors
- 2 duplicates
- Finding one error does not require the algorithm to stop

Semantic Analyzer

```scala
abstract class Exp {
  var pos: Position = _
}
def analyze(exp: Exp)(env: Env): Unit = exp match {
  case Lit(x) => ()
  case Unary(op, v) =>
    if (!isUnOperator(op)) error("undefined operator", exp.pos)
    analyze(v)(env)
  case Let(x, v, b) =>
    analyze(v)(env)
    analyze(b)(env.withVal(x))
  // ...
}
```

Let's Add Branches - Syntax

Let's Add Branches - Syntax

```
<op>    ::= ['*' | '/' | '+' | '-' | '<' | '>' | '=' | '!']+
<bop>   ::= ('<' | '>' | '=' | '!')[<op>]
<atom>  ::= <number>
          | '('<simp>')'
          | <ident>
          | '{'<exp>'}'
<uatom> ::= [<op>]<atom>
<cond>  ::= <simp><bop><simp>
<simp>  ::= <uatom>[<op><uatom>]*
          | 'if' '('<cond>')' <simp> 'else' <simp>
<exp>   ::= <simp>
          | 'val' <ident> '=' <simp>';' <exp>
```

Let’s Add Branches - AST

Let's Add Branches - AST

```
case class Cond(op: String, lop: Exp, rop: Exp) extends Exp
case class If(cond: Cond, tBranch: Exp, eBranch: Exp) extends Exp
```

Let's Add Branches - Semantics

Let’s Add Branches - Semantics

```scala
type Val = Int
def evalCond(op: String)(v: Val, w: Val) = op match {
  case "==" => v == w
  // ...
}
def eval(exp: Exp)(env: Env): Val = exp match {
  case If(Cond(op, l, r), tBranch, eBranch) =>
    if (evalCond(op)(eval(l)(env), eval(r)(env)))
      eval(tBranch)(env)
    else
      eval(eBranch)(env)
}
```

Let’s Add Branches - Semantics

```
type Val = Int
def evalCond(op: String)(v: Val, w: Val) = op match {
  case "==" => v == w
  // ...
}
def eval(exp: Exp)(env: Env): Val = exp match {
  case If(Cond(op, l, r), tBranch, eBranch) =>
    if (evalCond(op)(eval(l)(env), eval(r)(env)))
      eval(tBranch)(env)
    else
      eval(eBranch)(env)
}

eval(Let("x", 1,   // Omitted Lit
       If(Cond(">", Ref("x"), 0), Prim("+", 2, Ref("x")), 0)))(Map())
```

A Stack-Based Interpreter

The main idea is to be as close as possible to a processor. How does x86 handle branches?

A Stack-Based Interpreter

The main idea is to be as close as possible to a processor. How does x86 handle branches?

It is using flags. There are some instructions doing the comparison operations and setting the flags, and other instructions that jump to a code location depending on the value of the flags.

A Stack-Based Interpreter

The main idea is to be as close as possible to a processor. How does x86 handle branches?

It is using flags. There are some instructions doing the comparison operations and setting the flags, and other instructions that jump to a code location depending on the value of the flags.

Unfortunately, in Scala we can not "jump" to a code location. We can only simulate part of the behavior.

A Stack-Based Interpreter

```
val memory = new Array[Int](MEM_SIZE)
var flag = true
```

A Stack-Based Interpreter

```
val memory = new Array[Int](MEM_SIZE)
var flag = true

def evalCond(op: String)(sp: Int, sp1: Int) = op match {
  case "==" => flag = memory(sp) == memory(sp1)
  // ...
}
```

A Stack-Based Interpreter

A Stack-Based Interpreter

```scala
def eval(exp: Exp, sp: Int)(env: Env): Unit = exp match {
  case Cond(op, l, r) =>
    eval(l, sp)(env); eval(r, sp + 1)(env)
    evalCond(op)(sp, sp + 1)
  case If(cond, tBranch, eBranch) =>
    eval(cond, sp)(env)
    if (flag)
      eval(tBranch, sp)(env)
    else
      eval(eBranch, sp)(env)
}
```

A Stack-Based Interpreter

```
def eval(exp: Exp, sp: Int)(env: Env): Unit = exp match {
  case Cond(op, l, r) =>
    eval(l, sp)(env); eval(r, sp + 1)(env)
    evalCond(op)(sp, sp + 1)
  case If(cond, tBranch, eBranch) =>
    eval(cond, sp)(env)
    if (flag)
      eval(tBranch, sp)(env)
    else
      eval(eBranch, sp)(env)
}

eval(Let("x", 1,   // Omitted Lit
       If(Cond(">", Ref("x"), 0), Prim("+", 2, Ref("x")), 0)), 0)(Map())
```

A Stack-Based Compiler

```scala
def emitCode(exp: Exp): Unit = {
  emitln("val memory = new Array[Int](1000)")
  emitln("var flag = true")
  trans(exp, 0)(Env())
  emitln(s"memory(0)")
}
```

A Stack-Based Compiler

```
def emitCode(exp: Exp): Unit = {
  emitln("val memory = new Array[Int](1000)")
  emitln("var flag = true")
  trans(exp, 0)(Env())
  emitln(s"memory(0)")
}

def transCond(op: String)(sp: Loc, sp1: Loc) = op match {
  case "==" => emitln(s"flag = (memory($sp) == memory($sp1))")
  // ...
}
```

A Stack-Based Compiler

```scala
def trans(exp: Exp, sp: Int)(env: Env) = exp match {
  case Cond(op, l, r) =>
    trans(l, sp)(env); trans(r, sp+1)(env)
    transCond(op)(sp, sp + 1)
  case If(cond, tBranch, eBranch) =>
    trans(cond, sp)(env) // Set flag value
    emitln(s"if (flag) {")
    trans(tBranch, sp)(env)
    emitln("} else {")
    trans(eBranch, sp)(env)
    emitln("}")
}
```

A Stack-Based Compiler - Demo

```
emitCode(Let("x", 1,   // Omitted Lit
        If(Cond(">", Ref("x"), 0), Prim("+", 2, Ref("x")), 0)))
```

A Stack-Based Compiler - Demo

```
emitCode(Let("x", 1,   // Omitted Lit
       If(Cond(">", Ref("x"), 0), Prim("+", 2, Ref("x")), 0)))

val memory = new Array[Int](1000); var flag = true

memory(0) = 1
memory(1) = memory(0); memory(2) = 0
flag = (memory(1) >  memory(2))
if (flag) {
  memory(1) = 2; memory(2) = memory(0); memory(1) += memory(2)
} else {
  memory(1) = 0
}
memory(0) = memory(1)
memory(0)
```

x86 Flags And Jump

The x86 processor is using flags to handle comparison.

```
cmpq %rbx, %rax    # compare %rax to rbx and set the flags accordingly
```

x86 Flags And Jump

The x86 processor is using flags to handle comparison.

```
cmpq %rbx, %rax    # compare %rax to rbx and set the flags accordingly
```

Several instructions can be used for jump:

```
je  labelA         # jump equals
jne labelA         # jump not equals
jg  labelA         # jump greater
jge labelA         # jump greater or equals
jl  labelA         # jump less
jle labelA         # jump less or equals
jmp labelA         # always jump
```

x86 Flags And Jump - Example

```
    movq $0, %rax
    movq $1, %rbx
    cmpq %rbx, %rax     # operands inverted!!
    jg greater
    movq $1, %rax
greater:
    ret                 # what value is in %rax?
```

x86 Flags And Jump - Example

```
  movq $0, %rax
  movq $1, %rbx
  cmpq %rbx, %rax    # operands inverted!!
  jg greater
  movq $1, %rax
greater:
  ret                # what value is in %rax?
```

Returns 1, because 0 is not greater than 1 so the jump doesn't happen.

x86 Flags And Jump - Compile Ifs

```
trans(If(Cond(op, l, r), tBranch, eBranch), 0)(Map())
```

x86 Flags And Jump - Compile Ifs

```
trans(If(Cond(op, l, r), tBranch, eBranch), 0)(Map())
```

In order to compile the if statement, we are going to follow this idea:

```
  ... # code for l and r
  cmpq <r>, <l>
  j<op> if_then # the jump operation depends on 'op'
  ... #  code for else branch
  jmp if_end   # unconditional jump
if_then: # label for the beginning of the then branch
  ..  # code for then branch
if_end:
   ... # code for the rest
```

x86 Flags And Jump - Compile Ifs - Example

```
trans(If(Cond("==", 1, 0), 2, 3), 0)(Map())
```

x86 Flags And Jump - Compile Ifs - Example

```
trans(If(Cond("==", 1, 0), 2, 3), 0)(Map())

# begin code generated
  movq $1, %rbx # generate code that compute l, stored in %rbx
  movq $0, %rcx # generate code that compute r, stored in %rcx
  cmpq %rcx, %rbx
  je if1_then
  movq $3, %rbx # generate code for eBranch, store result in %rbx
  jmp if1_end
if1_then:
  movq $2, %rbx # generate code for tBranch, store result in %rbx
if1_end:
# end code generated
  movq %rbx, %rax
  ret
```

Where Are We?

We looked a little bit deeper in error handling. We made distinction between syntax errors and semantic errors. We also saw how to make our compiler more human friendly.

We added IF statements to our language and discussed the syntax, semantic and simple implementation.

Where Are We?

We looked a little bit deeper in error handling. We made distinction between syntax errors and semantic errors. We also saw how to make our compiler more human friendly.

We added IF statements to our language and discussed the syntax, semantic and simple implementation.

Questions?