

CS107: Error Handling, Semantics, and Branches

Guannan Wei

guannan.wei@tufts.edu

September 15, 2026

Spring 2026

Tufts University

- Project 1 dues today at 11:59 PM
- Project 2 will be released after the grace period
- Lab+office hour today 3:00-4:00 PM at Aidekman Arts Center, Room 012

- Parsing with generic operator precedence
- Tokenization
- Interpreting and compiling with let-bindings

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

We have been focusing on how to accept valid programs. One other aspect of the parser is to reject invalid programs.

What kind of error(s)?

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

Quiz

We have been focusing on how to accept valid programs. One other aspect of the parser is to reject invalid programs.

What kind of error(s)?

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

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

Quiz

We have been focusing on how to accept valid programs. One other aspect of the parser is to reject invalid programs.

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

We have been focusing on how to accept valid programs. One other aspect of the parser is to reject invalid programs.

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 the interpreter from previous lectures. We could choose to forbid redefining variables.

- 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, the compiler should generate correct code!

Token Position Information

```
// 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)

- What to do when we find a syntax error?

- What to do when we find a syntax error?

Different solutions:

- Report the error and exit

- What to do when we find a syntax error?

Different solutions:

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

- 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.

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

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

After **val** we expect an identifier and find a number. We can report an error, then create a “dummy” identifier and continue parsing.

```
val dummy = 2; 4
```

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

After **val** we expect an identifier and find a number. We can report an error, then create a “dummy” identifier and continue parsing.

```
val dummy = 2; 4
```

Other algorithms: Burke-Fisher error repair

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

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

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

```
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

- We want to find some semantic errors before code generation.

```
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))  
  // ...
```

- Like an interpreter, but does not deal with concrete values.
- We will see more about it in the next lecture.

Let's Add Branches - Syntax

Our language still misses many important constructs. Let's add branches!

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

- < simp > computes a value, but < exp > can represent a sequence of bindings/statements.
- **Exercise:** write down a well-formed expression using branches.

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 - AST

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

Roadmap:

- High-level value interpreter
- Stack-based interpreter simulating memory
- Stack-based compiler
- x86 code generation

Let's Add Branches - Semantics

```
type Val = Int
```

```
def evalCond(op: String)(v: Val, w: Val): Boolean = op match  
  case "==" => v == w  
  // ...
```

```
def eval(exp: Exp)(env: Env): Val = exp match  
  // other cases omitted  
  case If(Cond(op, l, r), tBranch, eBranch) =>  
    val lv = eval(l)(env)  
    val rv = eval(r)(env)  
    val cv = evalCond(op)(lv, rv)  
    if (cv)  
      eval(tBranch)(env)  
    else  
      eval(eBranch)(env)
```

Let's Add Branches - Semantics

Example:

Source program: `val x = 1; if (x > 0) 2 + x else 0`

Let's Add Branches - Semantics

Example:

Source program: `val x = 1; if (x > 0) 2 + x else 0`

AST:

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

A Stack-Based Interpreter

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

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

- It is using **flags**. There are comparison instructions that set the flags, and other instructions that **jump** to a code location depending on the value of the flags.

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

- It is using **flags**. There are comparison instructions that set 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)(sp1: Int, sp2: Int) = op match
  case "==" => flag = memory(sp1) == memory(sp2)
  // ...
```

A Stack-Based Interpreter

```
def eval(exp: Exp, sp: Int)(env: Env): Unit = exp match
  // other cases omitted
  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
  // other cases omitted
  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)
```

Example: (Lit omitted)

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

- Generating high-level Scala stack-based code:

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

- Generating high-level Scala stack-based code:

```
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)(sp1: Loc, sp2: Loc) = op match  
  case "==" => emitln(s"flag = (memory($sp1) == memory($sp2))")  
  // ...
```

A Stack-Based Compiler

```
def trans(exp: Exp, sp: Int)(env: Env) = exp match
  // other cases omitted
  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("if (flag) {")
    trans(tBranch, sp)(env)
    emitln("} else {")
    trans(eBranch, sp)(env)
    emitln("}")
```

A Stack-Based Compiler - Demo

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

A Stack-Based Compiler - Demo

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

Output:

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

A Stack-Based Compiler - Demo

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

Output:

```
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

X86 processors use **flags** to handle comparison.

```
cmpq %rbx, %rax    # compute %rax-%rbx and set the flags accordingly
```

x86 Flags And Jump

X86 processors use **flags** to handle comparison.

```
cmpq %rbx, %rax    # compute %rax-%rbx and set the flags accordingly
```

- Zero flag ZF = 1 if $\%rax - \%rbx = 0$
- Sign flag SF = 1 if $\%rax - \%rbx < 0$
- And other flags like overflow flag OF, carry flag CF, etc.

x86 Flags And Jump

X86 processors use **flags** to handle comparison.

```
cmpq %rbx, %rax    # compute %rax-%rbx and set the flags accordingly
```

- Zero flag ZF = 1 if $\%rax - \%rbx = 0$
- Sign flag SF = 1 if $\%rax - \%rbx < 0$
- And other flags like overflow flag OF, carry flag CF, etc.

Several instructions can be used for jump:

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

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.

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

```
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
```

Where Are We?

- Error handling: we made distinction between syntax errors and semantic errors.
- We also saw how to make our compiler more friendly to programmers.
- We added branches to our language, discussed its syntax, semantics, and code generation to x86.