CS107: Operator Precedence, Tokenization, and Bindings

Guannan Wei
guannan.wei@tufts.edu

Jan 20, 2026

Spring 2026

Tufts University

Recap: What did we learn last time?

Parsing with Operator Precedence

Parser from last time:

```
def parseExpression: Exp =
  var res = parseTerm
  while (in.hasNext(isOperator)) {
    in.next() match
      case '+' => res = Add(res, parseTerm)
      case '-' => res = Sub(res, parseTerm)
      case '*' => res = Mul(res, parseTerm)
      case '/' => res = Div(res, parseTerm)
  }
  res
```

Parsing with Operator Precedence

Parser from last time:

```
def parseExpression: Exp =
  var res = parseTerm
  while (in.hasNext(isOperator)) {
    in.next() match
      case '+' => res = Add(res, parseTerm)
      case '-' => res = Sub(res, parseTerm)
      case '*' => res = Mul(res, parseTerm)
      case '/' => res = Div(res, parseTerm)
  }
  res
```

Problem: Does not work for `1+2*3`

A Better Grammar

Grammar with explicit operator precedence:

```
<addop>  ::= '+' | '-'
<mulop>  ::= '*' | '/'
<factor> ::= <num>
<term>   ::= <factor>[<mulop><factor>]*
<exp>    ::= <term>[<addop><term>]*
```

A Better Grammar

Grammar with explicit operator precedence:

```
<addop>  ::= '+' | '-'
<mulop>  ::= '*' | '/'
<factor> ::= <num>
<term>   ::= <factor>[<mulop><factor>]*
<exp>    ::= <term>[<addop><term>]*
```

Example:

```
1+2*3 ∈ <exp>
  1 ∈ <term>
  + 2*3 ∈ <addop> <term>
    2 ∈ <factor>
    * 3 ∈ <mulop> <factor>
```

A Better Grammar

Grammar with explicit operator precedence:

```
<addop>  ::= '+' | '-'
<mulop>  ::= '*' | '/'
<factor> ::= <num>
<term>   ::= <factor>[<mulop><factor>]*
<exp>    ::= <term>[<addop><term>]*
```

Example:

```
1+2*3 ∈ <exp>
  1 ∈ <term>
  + 2*3 ∈ <addop> <term>
    2 ∈ <factor>
    * 3 ∈ <mulop> <factor>
```

Implementation in Project 1

Quiz

```
3*2/3 // => ???
```

Quiz

```
3*2/3 // => ???
3/2*3 // => ???
```

Quiz

```
3*2/3 // => ???

3/2*3 // => ???

3*5%4&4==4 // => ???
```

Quiz

```
3*2/3 // => ???
3/2*3 // => ???
3*5%4&4==4 // => ???
```

Answer:

- `(3*2)/3` => 2
- `(3/2)*3` => 3
- `((3*5)%4)&(4==4)` => 1 (in C), type error in Scala
  - weak typing vs. strong typing

Generic Operator Precedence

We define a grammar that can be easily adapted with new operators:

```
<op>  ::= '+' | '-' | '*' | '/'
<exp> ::= <num>[<op><num>]*
```

Generic Operator Precedence

We define a grammar that can be easily adapted with new operators:

```
<op>  ::= '+' | '-' | '*' | '/'
<exp> ::= <num>[<op><num>]*
```

In addition, we define a **precedence** level for each operator. From low to high:

- `'+'`, `'-'`
- `'*'`, `'/'`

Generic Operator Precedence

We define a grammar that can be easily adapted with new operators:

```
<op>  ::= '+' | '-' | '*' | '/'
<exp> ::= <num>[<op><num>]*
```

In addition, we define a **precedence** level for each operator. From low to high:

- `'+'`, `'-'`
- `'*'`, `'/'`

We also need to define the **associativity** of each operator.

```
2 + 3 * 4 => 2 + (3 * 4) // * precedes +
2 * 3 / 4 => (2 * 3) / 4 // left associative
x = y = z => x = (y = z) // right associative
```

Code Update

At the same time, we make our target language more general with a new AST node:

```
case class Prim(op: Char, a: Exp, b: Exp) extends Exp
```

And we encode the precedence level with a function:

```
def prec(op: Char) = op match
  case '+' | '-' => 0
  case '*' | '/' => 1

def isInfixOp(min: Int)(op: Char) = prec(op) >= min
```

Now We Can Parse

```
def parseNum: Exp = Lit(getNum)

def parseExpression: Exp = parseExpression(0)
def parseExpression(min: Int): Exp =
  var res = parseNum
  while (in.hasNext(...)) {
    val op = getOperator
    ...
  }
  res
```

Now We Can Parse

```
def parseNum: Exp = Lit(getNum)

def parseExpression: Exp = parseExpression(0)
def parseExpression(min: Int): Exp =
  var res = parseNum
  while (in.hasNext(...)) {
    val op = getOperator
    ...
  }
  res
```

Let's try it by hand:

- `1+2*3`
- `2*3/4`

Now We Can Parse

Final solution in Project 2

Let's Introduce Immutable Variables

```
<op>    ::= '+' | '-' | '*' | '/'
<atom>  ::= <num>
          | <ident>
          | '('<exp>')'
<exp>   ::= <atom>[<op><atom>]*
          | 'val' <ident> '=' <exp>';' <exp>
```

Let's Introduce Immutable Variables

```
<op>    ::= '+' | '-' | '*' | '/'
<atom>  ::= <num>
          | <ident>
          | '('<exp>')'
<exp>   ::= <atom>[<op><atom>]*
          | 'val' <ident> '=' <exp>';' <exp>
```

Example:

```
val x = 10;
x*2
```

Let's Introduce Immutable Variables

```
<op>    ::= '+' | '-' | '*' | '/'
<atom>  ::= <num>
          | <ident>
          | '('<exp>')'
<exp>   ::= <atom>[<op><atom>]*
          | 'val' <ident> '=' <exp>';' <exp>
```

Example:

```
val x = 10;
x*2
```

New AST representations:

```
case class Ref(name: String) extends Exp
case class Let(name: String, rhs: Exp, body: Exp) extends Exp
```

Interpreter

```
type Val = Int

def eval(exp: Exp): Val = exp match
  // previous cases omitted ...
  case Let(n, rhs, b) =>
    val ev = eval(rhs)
    val eb = eval(b)
    ???
  case Ref(name) => ???
```

What can we do?

Interpreter

Idea: use an environment to track variable bindings:

```
type Val = Int

def eval(exp: Exp)(env: Map[String, Val]): Val = exp match
  // previous cases ...
  case Let(n, rhs, b) => ???
  case Ref(name) => ???
```

Interpreter

Idea: use an environment to track variable bindings:

```
type Val = Int

def eval(exp: Exp)(env: Map[String, Val]): Val = exp match
  // previous cases ...
  case Let(n, rhs, b) => ???
  case Ref(name) => ???
```

Example:

```
eval(Let("x", Lit(10), Prim("*", Ref("x"), Lit(2))))(Map()) // => ???
```

Interpreter

```
type Val = Int

def eval(exp: Exp)(env: Map[String, Val]): Val = exp match
  // previous cases omitted ...
  case Let(n, rhs, b) =>
    eval(b)(env + (n -> eval(rhs)(env)))
  case Ref(n) =>
    env(n)
```

Interpreter

```
type Val = Int

def eval(exp: Exp)(env: Map[String, Val]): Val = exp match
  // previous cases omitted ...
  case Let(n, rhs, b) =>
    eval(b)(env + (n -> eval(rhs)(env)))
  case Ref(n) =>
    env(n)
```

Example:

```
eval(Let("x", Lit(10), Prim("*", Ref("x"), Lit(2))))(Map()) // => 20
```

A Stack-Based Interpreter

Idea: environment tracks variable locations in memory (instead of values).

```
type Val = Int
val memory = new Array[Int](MEM_SIZE)

def eval(exp: Exp, sp: Int)(env: Map[String, Int]): Unit = exp match
  // previous cases omitted ...
  case Let(n, rhs, b) =>
    eval(rhs, sp)(env)
    eval(b, sp + 1)(env + (n -> sp))
    memory(sp) = memory(sp + 1)
  case Ref(n) =>
    memory(sp) = memory(env(n))
```

A Stack-Based Interpreter

Idea: environment tracks variable locations in memory (instead of values).

```
type Val = Int
val memory = new Array[Int](MEM_SIZE)

def eval(exp: Exp, sp: Int)(env: Map[String, Int]): Unit = exp match
  // previous cases omitted ...
  case Let(n, rhs, b) =>
    eval(rhs, sp)(env)
    eval(b, sp + 1)(env + (n -> sp))
    memory(sp) = memory(sp + 1)
  case Ref(n) =>
    memory(sp) = memory(env(n))
```

Example:

```
eval(Let("x", Lit(10), Prim("*", Ref("x"), Lit(2))))(Map()) // => 20
```

A Stack-Based Compiler

```
def trans(exp: Exp, sp: Int)(env: Map[String, Int]): Unit = exp match
  // ...
  case Let(n, rhs, b) =>
    trans(rhs, sp)(env)
    trans(b, sp + 1)(env + (n -> sp))
    println(s"memory($sp) = memory(${sp + 1})")
  case Ref(n) => println(s"memory($sp) = memory(${env(n)})")

// val x = 10; x*2
trans(Let("x", Lit(10), Prim("*", Ref("x"), Lit(2))))(...)
```

A Stack-Based Compiler

```
def trans(exp: Exp, sp: Int)(env: Map[String, Int]): Unit = exp match
  // ...
  case Let(n, rhs, b) =>
    trans(rhs, sp)(env)
    trans(b, sp + 1)(env + (n -> sp))
    println(s"memory($sp) = memory(${sp + 1})")
  case Ref(n) => println(s"memory($sp) = memory(${env(n)})")

// val x = 10; x*2
trans(Let("x", Lit(10), Prim("*", Ref("x"), Lit(2))))(...)
```

Expected output:

```
memory(0) = 10          // Lit(10), Map()
memory(1) = memory(0)   // Ref("x"), Map("x" -> 0)
memory(2) = 2           // Lit(2), Map("x" -> 0)
memory(1) *= memory(2)  // Prim("*", ...), Map("x" -> 0)
memory(0) = memory(1)   // Let("x", ...), Map()
```

A Stack-Based Compiler Targeting x86-64 Registers

- Using machine registers:

```
val regs = Seq("%rbx", "%rcx", "%rdi", "%rsi", "%r8", "%r9")
def trans(exp: Exp, sp: Int)(env: Map[String, Int]): Unit = exp match
  // ...
  case Let(n, rhs, b) =>
    trans(rhs, sp)(env)
    trans(b, sp + 1)(env + (n -> sp))
    println(s"movq ${regs(sp + 1)}, ${regs(sp)}")
  case Ref(n) => println(s"movq ${regs(env(n))}, ${regs(sp)})")

// val x = 10; x*2
trans(Let("x", Lit(10), Prim("*", Ref("x"), Lit(2))))(...)

movq $10, %rbx
movq %rbx, %rcx
movq $2, %rdi
imulq %rdi, %rcx
movq %rcx, %rbx
```

Parsing

```
val x = 10;
x*2
```

Can we parse this expression?

Parsing

```
val x = 10;
x*2
```

Can we parse this expression?

Not with the strategy we were using. Let's introduce tokenization.

Tokenization: convert the stream of characters to a stream of tokens/words, performed by tokenizers/scanners/lexers.

Token: a meaningful unit in the source code, such as keywords, identifiers, operators, literals, and delimiters.

Tokenization

```
val x = 10;
x*2
```

What tokens do we have?

- Numbers: `[0-9]+`

Tokenization

```
val x = 10;
x*2
```

What tokens do we have?

- Numbers: `[0-9]+`
- Keywords: `val`

Tokenization

```
val x = 10;
x*2
```

What tokens do we have?

- Numbers: `[0-9]+`
- Keywords: `val`
- Identifier: `[a-zA-Z][a-zA-Z0-9]*`

Tokenization

```
val x = 10;
x*2
```

What tokens do we have?

- Numbers: `[0-9]+`
- Keywords: `val`
- Identifier: `[a-zA-Z][a-zA-Z0-9]*`
- Delimiters: `'='`, `';'`

Tokenization

```
val x = 10;
x*2
```

What tokens do we have?

- Numbers: `[0-9]+`
- Keywords: `val`
- Identifier: `[a-zA-Z][a-zA-Z0-9]*`
- Delimiters: `'='`, `';'`

Tokenization

```
val x = 10;
x*2
```

What tokens do we have?

- Numbers: `[0-9]+`
- Keywords: `val`
- Identifier: `[a-zA-Z][a-zA-Z0-9]*`
- Delimiters: `'='`, `';'`

And we ignore whitespace: `' '`, `'\n'`, `'\r'`, `'\t'`

Scanner

- Defining the Tokens:

```
abstract class Token
case object EOF extends Token
case class Number(x: Int) extends Token
case class Ident(x: String) extends Token
case class Keyword(x: String) extends Token
case class Delim(x: Char) extends Token
```

Scanner

- Defining the Tokens:

```
abstract class Token
case object EOF extends Token
case class Number(x: Int) extends Token
case class Ident(x: String) extends Token
case class Keyword(x: String) extends Token
case class Delim(x: Char) extends Token
```

- Note that `Scanner` is a `Reader [ Token ]`, which is a stream of tokens:

```
class Scanner(in: Reader[Char]) extends Reader[Token]
```

Scanner

- Some primitive predicates to classify characters:

```
def isAlpha(c: Char) =
  ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z')

def isDigit(c: Char) = '0' <= c && c <= '9'

def isAlphaNum(c: Char) = isAlpha(c) || isDigit(c)

val isWhiteSpace = Set(' ','\t','\n','\r')

val isOperator   = Set('+','-','*','/')

val isDelim      = Set('(',')',';','=')

val isKeyword    = Set("val")
```

- Note: in Scala, `s(x)` tests membership $x \in s$ for a set `s` (`apply` method).

Scanner

```
def getToken(): Token =
  while (in.hasNext(isWhiteSpace)) in.next() // skip white space
  if (in.hasNext(isAlpha))
    getName()
  else if (in.hasNext(isOperator))
    getOperator()
  else if (in.hasNext(isDigit))
    getNum()
  else if (in.hasNext(isDelim))
    Delim(in.next())
  else if (!in.hasNext)
    EOF
  else
    abort(s"Unexpected character: ${in.peek}")
```

More Than One Letter!

```
def getName() =
  val buf = new StringBuilder
  while (in.hasNext(isAlphaNum)) {
    buf += in.next()
  }
  val s = buf.toString
  if (isKeyword(s)) Keyword(s) else Ident(s)
```

We need to distinguish between keywords and identifiers.

```
val x = 10;
x*2
```

More than just a digit!

Previously:

```
def getNum =
  if (in.hasNext(isDigit)) (in.next - '0')
```

More than just a digit!

Previously:

```
def getNum =
  if (in.hasNext(isDigit)) (in.next - '0')
```

Now:

```
def getNum =
  var res = 0
  while (in.hasNext(isDigit))
    res = res * 10 + (in.next - '0')
  Number(res)
```

Tokenization + Parsing

```
class Scanner(in: Reader[Char]) extends Reader[Token] {
  def peek = ...
  def next = ...
  def hasNext(f: Token => Boolean) = f(peek)
  ...
}
class Parser(in: Reader[Token]) {
  def parseExpression: Exp = ...
  ...
}
```

Look good?

Error Handling

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

Does our current parser achieve that goal perfectly?

What can be added to improve the error handling?

Error Handling

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

Does our current parser achieve that goal perfectly?

What can be added to improve the error handling?

- Report more information. Such as: line number, give some hints

Error Handling

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

Does our current parser achieve that goal perfectly?

What can be added to improve the error handling?

- Report more information. Such as: line number, give some hints
- Try to recover and continue parsing

Error Handling

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

Does our current parser achieve that goal perfectly?

What can be added to improve the error handling?

- Report more information. Such as: line number, give some hints
- Try to recover and continue parsing
- Test integer overflow (for literals)

Error Handling

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

Does our current parser achieve that goal perfectly?

What can be added to improve the error handling?

- Report more information. Such as: line number, give some hints
- Try to recover and continue parsing
- Test integer overflow (for literals)
- Think about it for the next lecture

Where are we?

- Generic operator precedence handling.
- Immutable variables and let-bindings.
  - Interpreter and compiler implementations.
- Improved the parser to process words (tokens) instead of individual characters.
  - Pipeline: source code -> tokens -> AST -> interpreter/compiler

Where are we?

- Generic operator precedence handling.
- Immutable variables and let-bindings.
  - Interpreter and compiler implementations.
- Improved the parser to process words (tokens) instead of individual characters.
  - Pipeline: source code -> tokens -> AST -> interpreter/compiler

Questions?