PURDUE
UNIVERSITY

Operator precedence and Tokenization

Compilers: Principles And Practice

Tiark Rompf

What did we learn in the last class?

Return to Operator Precedence

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

Return to Operator Precedence

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

Does not work for 1+2*3

A Better Grammar

A small reminder of what we have so far:

```
<exp> ::= <num>
        |  <exp> + <exp>
        |  <exp> - <exp>
        |  <exp> * <exp>
        |  <exp> / <exp>
```

A Better Grammar

A small reminder of what we have so far:

```
<exp> ::= <num>
        |  <exp> + <exp>
        |  <exp> - <exp>
        |  <exp> * <exp>
        |  <exp> / <exp>
```

Defines **legal sentences**, but does not define **how to parse them**

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>]*
```

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), error in Scala

Generic Operator Precedence

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

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

Generic Operator Precedence

We define a grammar that can be easily updated 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 updated 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 // left associative
x = y = z => x = (y = z) // right associative
```

Code Update

At the same time we also make our target language more general.

```
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>

val x = 10;
x*2
```

Let's Introduce Immutable Variables

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

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

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

Interpreter

```
type Val = Int

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

Interpreter

```
type Val = Int

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

What can we do?

Interpreter

```
type Val = Int

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

Interpreter

```
type Val = Int

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

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

Interpreter

```
type Val = Int

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

Interpreter

```
type Val = Int

def eval(exp: Exp)(ctx: Map[String,Val]): Val = exp match {
  // previous cases
  // ...
  case Let(n, v, b) =>
    eval(b)(ctx + n -> eval(v)(ctx))
  case Ref(n) =>
    ctx(n)
}

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

A Stack-Based Interpreter

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

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

A Stack-Based Interpreter

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

def eval(exp: Exp, sp: Int)(ctx: Map[String,Int]): Unit = exp match {
  // previous cases
  // ...
  case Let(n, v, b) =>
    eval(v, sp)(ctx)
    eval(b, sp + 1)(ctx + n -> sp)
    memory(sp) = memory(sp + 1)
  case Ref(n) => memory(ctx(n))
}

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

A Stack-Based Compiler

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


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

A Stack-Based Compiler

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

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

```
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

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

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

```
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

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

Scanner

```
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")
```

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!

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

More than just a digit!

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

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

More than just a digit!

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

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

Look good?

Error Handling

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

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 program as well.

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 program as well.

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

Error Handling

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

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

Error Handling

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

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

Think about it for the next lecture

Where are we?

We have seen a technique to handle generic operator precedence.

We added immutable variables to our language and we improved the parser to handle tokens.

Where are we?

We have seen a technique to handle generic operator precedence.

We added immutable variables to our language and we improved the parser to handle tokens.

Questions?

Have a good weekend!