Recapitulation from Previous Lectures

Recap: Parser Implementation

Use of ***recursive descent***:

mutually-recursive methods corresponding to each production of a grammar

For *easy* implementation, the grammar should be LL(1)

NULLABLE, *FIRST*, *FOLLOW* allow us know what to do next

We build ***abstract syntax trees*** from the parser input

Precedence, *associativity* of operators: tricky and cumbersome...

Recap: Example Language

```
// lexing: "A + B * C" ⇒ Ident("A"),Plus,Ident("B"),Times, ...

enum Token:
  case Ident(name: String)
  case OpenParen
  case CloseParen
  case Plus
  case Times
```

Recap: Example Language

```
// lexing: "A + B * C" ⇒ Ident("A"),Plus,Ident("B"),Times, ...

enum Token:
  case Ident(name: String)
  case OpenParen
  case CloseParen
  case Plus
  case Times

// parsing: ... ⇒ Add( Var("A") , Mult(Var("B"), Var("C")) )

enum Expr:
  case Var(name: String)
  case Add(lhs: Expr, rhs: Expr)
  case Mult(lhs: Expr, rhs: Expr)
```

Recap: Mutable Parser Architecture

```
class Parser(ite: Iterator[Token]):

  var cur: Option[Token] = ite.nextOption
```

Recap: Mutable Parser Architecture

```
class Parser(ite: Iterator[Token]):

  var cur: Option[Token] = ite.nextOption
  def consume: Unit = { cur = ite.nextOption }
  def skip(tk: Token): Unit =
    if cur != Some(tk)
        then fail("expected " + tk + ", found " + cur)
    consume
```

Recap: Mutable Parser Architecture

```
class Parser(ite: Iterator[Token]):

  var cur: Option[Token] = ite.nextOption
  def consume: Unit = { cur = ite.nextOption }
  def skip(tk: Token): Unit =
    if cur != Some(tk)
        then fail("expected " + tk + ", found " + cur)
    consume

  // define parser:
  def expr = ... atom ...
  def atom: Expr = cur match
    case Some(Ident(nme)) ⇒ consume; Var(nme)
    case OpenParen ⇒ consume;
      val e = expr; skip(CloseParen); e
    case _ ⇒ fail("expected atomic expression, found " + cur)
```

Recap: Implementing Precedence and Associativity Right

Idea: make operator-parsing methods return *lists*,

then *fold* these into ASTs with correct associativity (*foldLeft*, *foldRight*).

```
def atom = ... // as before
def multiplyAtoms: List[Expr] = ???  // parses:  * atom * atom * ...
def product: Expr = ???              // parses:  atom * atom * ...
def addProducts: List[Expr] = ???    // parses:  + prod + prod + ...
def expr: Expr = ???                 // parses:  prod + prod + ...
```

Pratt Parsing

Pratt Parsing: Motivation

Irritating questions about parsing operators:

- How to avoid having to manually transform grammars?

  We'd like to separately specify operator precedence/associativity.

Pratt Parsing: Motivation

Irritating questions about parsing operators:

- How to avoid having to manually transform grammars?

  We'd like to separately specify operator precedence/associativity.

- How to support user-defined operators and parse them correctly?

Pratt Parsing: Idea

Simplest way of describing precedence and associativity:

operators have *distinct **left** and **right** precedences*

E.g., '$+$' has (3, 4) and '*' has (5, 6)

Pratt Parsing: Idea

Simplest way of describing precedence and associativity:

operators have *distinct* ***left*** *and* ***right*** *precedences*

E.g., ‘$+$’ has (3, 4) and ‘*’ has (5, 6)

Precedence determines which side “binds stronger”

tokens	A		+		B		*		C
precedences		3		4		5		6	

Pratt Parsing: Idea

Simplest way of describing precedence and associativity:

operators have *distinct **left** and **right** precedences*

E.g., ‘$+$’ has (3, 4) and ‘*’ has (5, 6)

Precedence determines which side “binds stronger”

tokens	A		$+$		B		*		C
precedences		3		4 $\not\leftarrow$		$\rightarrow$ 5		6	

Pratt Parsing: Idea

Simplest way of describing precedence and associativity:

operators have *distinct* ***left*** *and* ***right*** *precedences*

E.g., '+' has (3, 4) and '*' has (5, 6)

Precedence determines which side "binds stronger"

tokens	A		+		B		*		C
precedences		3		4	$\not\leftarrow \quad \longrightarrow$	5		6	

Subsumes notion of associativity!

tokens	A		+		B		+		C
precedences		3		4		3		4	

Pratt Parsing: Idea

Simplest way of describing precedence and associativity:

operators have *distinct* ***left*** *and* ***right*** *precedences*

E.g., '$+$' has (3, 4) and '*' has (5, 6)

Precedence determines which side "binds stronger"

tokens	A		$+$		B		*		C
precedences		3		4	$\not\leftarrow\ \ \rightarrow$	5		6	

Subsumes notion of associativity!

tokens	A		$+$		B		$+$		C
precedences		3		4	$\leftarrow\ \ \not\rightarrow$	3		4	

Pratt Parsing: Implementation (1)

Defining the precedences:

```
def opPrec(opStr: String): (Int, Int)
```

Pratt Parsing: Implementation (1)

Defining the precedences:

```
def opPrec(opStr: String): (Int, Int) = opStr match
  case "*" ⇒ (50, 51)
```

Pratt Parsing: Implementation (1)

Defining the precedences:

```
def opPrec(opStr: String): (Int, Int) = opStr match
  case "*" ⇒ (50, 51)
  case "+" ⇒ (30, 31)
  case "⇒" ⇒ (21, 20)
  case ...
```

Pratt Parsing: Implementation (1)

Defining the precedences:

```
def opPrec(opStr: String): (Int, Int) = opStr match
  case "*" ⇒ (50, 51)
  case "+" ⇒ (30, 31)
  case "⇒" ⇒ (21, 20)
  case ...
```

Example language. Tokens and abstract syntax:

```
enum Token { case OpenParen; case CloseParen
  case Ident(name: String); case Oper(name: String) }
```

Pratt Parsing: Implementation (1)

Defining the precedences:

```
def opPrec(opStr: String): (Int, Int) = opStr match
  case "*" ⇒ (50, 51)
  case "+" ⇒ (30, 31)
  case "⇒" ⇒ (21, 20)
  case ...
```

Example language. Tokens and abstract syntax:

```
enum Token { case OpenParen; case CloseParen
  case Ident(name: String); case Oper(name: String) }

enum Expr { case Var(name: String)
  case Infix(lhs: Expr, op: String, rhs: Expr) }
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
  case _ ⇒ fail(rest)
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
  case _ ⇒ fail(rest)

// Having parsed acc, what to do next at this precedence?
def exprCont(acc: Expr, prec: Int): Expr
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
  case _ ⇒ fail(rest)

// Having parsed acc, what to do next at this precedence?
def exprCont(acc: Expr, prec: Int): Expr = cur match
  case Some(Oper(opStr))
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
  case _ ⇒ fail(rest)

// Having parsed acc, what to do next at this precedence?
def exprCont(acc: Expr, prec: Int): Expr = cur match
  case Some(Oper(opStr)) if opPrec(opStr)._1 > prec ⇒
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
  case _ ⇒ fail(rest)

// Having parsed acc, what to do next at this precedence?
def exprCont(acc: Expr, prec: Int): Expr = cur match
  case Some(Oper(opStr)) if opPrec(opStr)._1 > prec ⇒
    consume
    val rhs = expr(opPrec(opStr)._2)
    exprCont(Infix(acc, opStr, rhs), prec)
```

Pratt Parsing: Implementation (2)

```
def expr(prec: Int): Expr = cur match
  case Some(Ident(nme)) ⇒
    consume; exprCont(Var(nme), prec)
  case Some(OpenParen) ⇒
    consume; val res = expr(0); skip(CloseParen)
    exprCont(res, prec)
  case _ ⇒ fail(rest)

// Having parsed acc, what to do next at this precedence?
def exprCont(acc: Expr, prec: Int): Expr = cur match
  case Some(Oper(opStr)) if opPrec(opStr)._1 > prec ⇒
    consume
    val rhs = expr(opPrec(opStr)._2)
    exprCont(Infix(acc, opStr, rhs), prec)
  case _ ⇒ acc
```

Pratt Parsing: Example 1

tokens	expression	prec	acc
A,+,B,*,C	expr(0)	0	
+,B,*,C	consume	0	
+,B,*,C	exprCont(Var("A"), 0)	0	"A"
B,*,C	consume	0	"A"
B,*,C	expr(opPrec("+")._2)	4	
*,C	consume	4	
*,C	exprCont(Var("B"), 4)	4	"B"
C	consume	4	"B"
C	expr(opPrec("*")._2)	6	
	consume	6	
	exprCont(Var("C"), 6)	6	"C"
	acc	6	"C"
	exprCont(Infix(acc, "*", rhs), 6)	6	("B" * "C")
	acc	6	("B" * "C")
	exprCont(Infix(acc, "+", rhs), 6)	6	("A" + ("B" * "C"))
	acc	6	("A" + ("B" * "C"))

Pratt Parsing: Example 2

tokens	expression	prec	acc
A,+,B,+,C	expr(0)	0	
+,B,+,C	consume	0	
+,B,+,C	exprCont(Var("A"), 0)	0	"A"
B,+,C	consume	0	"A"
B,+,C	expr(opPrec("+")._2)	4	
+,C	consume	4	
+,C	exprCont(Var("B"), 4)	4	"B"
+,C	acc	4	"B"
+,C	exprCont(Infix(acc, "+", rhs), 0)	0	("A" + "B")
C	consume	0	"B"
C	expr(opPrec("+")._2)	4	
	consume	4	
	exprCont(Var("C"), 0)	4	"C"
	acc	4	"C"
	exprCont(Infix(acc, "+", rhs), prec)	0	(("A" + "B") + "C")
	acc	0	(("A" + "B") + "C")

User-defined operators

Some programming languages allow users to ***define their own*** operators

User-defined operators

Some programming languages allow users to ***define their own*** operators

There are different design philosophies:

- Uniform rules of precedence and associativity

  Examples: Scala, OCaml

  Pro: predictable

User-defined operators

Some programming languages allow users to ***define their own*** operators

There are different design philosophies:

- Uniform rules of precedence and associativity

  Examples: Scala, OCaml

  Pro: predictable

- User-specified precedence and associativity for each operator

  Examples: Haskell, Coq

  Pro: very flexible

User-defined operators

Some programming languages allow users to ***define their own*** operators

There are different design philosophies:

- Uniform rules of precedence and associativity

  Examples: Scala, OCaml

  Pro: predictable

- User-specified precedence and associativity for each operator

  Examples: Haskell, Coq

  Pro: very flexible

While Haskell's approach is more flexible,

it makes ***reading*** *code* for people unfamiliar with your operators *a lot* ***harder***

Character Precedence Tables

Scala determines the *precedence* of an operator based on its ***first letter***

Character Precedence Tables

Scala determines the *precedence* of an operator based on its ***first letter***:

```
(all letters)
|
^
&
= !
< >
:
+ -
* / %
(all other special characters)
```

Scala operators are ***right-associative*** only when they *end with a **colon*** ':'

Q: What to do when mixing associativity with same precedence?

Character Precedence Tables

Scala determines the *precedence* of an operator based on its ***first letter***:

```
(all letters)
|
^
&
= !
< >
:
+ -
* / %
(all other special characters)
```

Scala operators are ***right-associative*** only when they *end with a* ***colon*** ‘:’

Q: What to do when mixing associativity with same precedence? $\Rightarrow$ report error

Example Character Precedence Table

```
val prec: Map[Char, Int] = """
  =
  @
  :
  |
  ^
  &
  !
  < >
  + -
  * / %
""".split('\n').zipWithIndex.flatMap { (cs, i) ⇒
    cs.filterNot(_.isWhitespace).map(_ -> i)
}.toMap.withDefaultValue(Int.MaxValue) // prec('~') == 2147483647
```

⇒ returns `Map('=' -> 0, '@' -> 1, ... , '<' -> 7, '>' -> 7, ... )`

A Nicer Approach?

Based on our Pratt parsing approach,

*why not use the **first** and **last** characters*

*to determine both **precedence** and **associativity**?*

A Nicer Approach?

Based on our Pratt parsing approach,

*why not use the **first** and **last** characters*

*to determine both **precedence** and **associativity**?*

Example:

"a + b |> f |> g" *parses as* "((a + b) |> f) |> g"

A Nicer Approach?

Based on our Pratt parsing approach,

*why not use the **first** and **last** characters*

*to determine both **precedence** and **associativity**?*

Example:

"a $+$ b $|>$ f $|>$ g" *parses as* "((a $+$ b) $|>$ f) $|>$ g"

Makes *symmetric operators* like $|>$ and $<|$ behave *symmetrically*

A Nicer Approach?

Based on our Pratt parsing approach,

*why not use the **first** and **last** characters*

*to determine both **precedence** and **associativity**?*

Example:

"a + b |> f |> g" *parses as* "((a + b) |> f) |> g"

Makes *symmetric operators* like |> and <| behave *symmetrically*

Determines *associativity* naturally:

"S –> T –> U" *parses as* "S –> (T –> U)"

A Nicer Approach?

Based on our Pratt parsing approach,

*why not use the **first** and **last** characters*

*to determine both **precedence** and **associativity**?*

Example:

"a + b |> f |> g" *parses as* "((a + b) |> f) |> g"

Makes *symmetric operators* like |> and <| behave *symmetrically*

Determines *associativity* naturally:

"S -> T -> U" *parses as* "S -> (T -> U)"

```
def opPrec(opStr: String): (Int, Int) =
      (prec(opStr.head), prec(opStr.last))
```

Parsing-Expression Grammars (PEG)

Parsing-Expression Grammars (PEG)

A more recent ***alternative*** to *context-free grammars* (CFG)

Parsing-Expression Grammars remove ambiguities through ***biased*** *choice*:

Instead of X | Y, use X / Y

which tries to parse Y ***only*** *if parsing X* ***fails***!

Parsing-Expression Grammars (PEG)

A more recent ***alternative*** to *context-free grammars* (CFG)

Parsing-Expression Grammars remove ambiguities through ***biased*** *choice*:

Instead of X | Y, use X / Y

which tries to parse Y ***only*** *if parsing X* ***fails***!

Example:

$$
\begin{aligned}
E &\rightarrow F + E \ / \ F \\
F &\rightarrow T * F \ / \ T \\
T &\rightarrow D^{\star} \ / \ \text{‘('} \ E \ \text{‘)'} \\
D &\rightarrow 0 \ / \ 1 \ / \ 2 \ / \ 3 \ / \ 4 \ / \ 5 \ / \ 6 \ / \ 7 \ / \ 8 \ / \ 9
\end{aligned}
$$

Parsing-Expression Grammars (PEG)

A more recent ***alternative*** to *context-free grammars* (CFG)

Parsing-Expression Grammars remove ambiguities through ***biased*** *choice*:

- Instead of X|Y, use X/Y
  - which tries to parse Y ***only*** *if parsing X* ***fails***!

Compare with LL(1) and related approaches, which

- detect ambiguities in grammar before parsing starts
- but may require awkward grammar modifications

Parsing-Expression Grammars (PEG)

A more recent ***alternative*** to *context-free grammars* (CFG)

Parsing-Expression Grammars remove ambiguities through ***biased*** *choice*:

- Instead of X|Y, use X/Y
  - which tries to parse Y ***only*** *if parsing X* ***fails***!

Compare with LL(1) and related approaches, which

- detect ambiguities in grammar before parsing starts
- but may require awkward grammar modifications

CFGs: more natural algebraic properties (e.g., disjunctions are commut. and assoc.)

Composition of PEGs is trickier & can be confusing

Parsing-Expression Grammars (PEG)

A more recent ***alternative*** to *context-free grammars* (CFG)

Parsing-Expression Grammars remove ambiguities through ***biased*** *choice*:

- Instead of X|Y, use X/Y
  - which tries to parse Y ***only*** *if parsing X* ***fails***!

Compare with LL(1) and related approaches, which

- detect ambiguities in grammar before parsing starts
- but may require awkward grammar modifications

CFGs: more natural algebraic properties (e.g., disjunctions are commut. and assoc.)

Composition of PEGs is trickier & can be confusing

However, PEGs can be more natural for programmers (parsing follows grammar)

- especially compared with advanced bottom up parsing algorithms like LR(1)

Packrat Parsing for PEGs

Guaranteed to *run in **linear** time* through heavy use of *memoisation*

Never raise ambiguities, but may loop indefinitely (left-recursion)

Packrat Parsing for PEGs

Guaranteed to *run in **linear** time* through heavy use of *memoisation*

Never raise ambiguities, but may loop indefinitely (left-recursion)

However, may require some debugging

if something does not parse following expectations

Packrat Parsing for PEGs

Guaranteed to *run in **linear** time* through heavy use of *memoisation*

Never raise ambiguities, but may loop indefinitely (left-recursion)

However, may require some debugging

if something does not parse following expectations

Packrat Parsing tends to be **slower** than many other linear-time parsing techniques

(Becket and Somogyi, 2008; Grimm, 2004).

Packrat Parsing for PEGs

Guaranteed to *run in **linear** time* through heavy use of *memoisation*

Never raise ambiguities, but may loop indefinitely (left-recursion)

However, may require some debugging

if something does not parse following expectations

Packrat Parsing tends to be **slower** than many other linear-time parsing techniques

(Becket and Somogyi, 2008; Grimm, 2004).

Some approaches can also handle left recursion (with worse complexity)

Popularity Packrat Parsing

Very ***popular*** in modern programming languages ecosystems

Can be *embedded* as a *domain-specific language* (DSL)

(no need for external code generation or the like)

Popularity Packrat Parsing

*Very **popular*** in modern programming languages ecosystems

Can be *embedded* as a *domain-specific language* (DSL)

(no need for external code generation or the like)

Examples

Scala libraries: fastparse, cats-parse, ...

Haskell libraries: Parsec, ...

etc.

Popularity Packrat Parsing

*Very **popular*** in modern programming languages ecosystems

Can be *embedded* as a *domain-specific language* (DSL)

(no need for external code generation or the like)

Examples

Scala libraries: fastparse, cats-parse, ...

Haskell libraries: Parsec, ...

etc.

However, the library you will use in this project, **Scallion**,

is based on **LL(1)** parsing (internally using derivatives), *not* Packrat