Recapitulation from Previous Lectures

Recap: Grammars

Name the productions of a language:

$$\begin{aligned} \mathit{start} &\rightarrow \mathit{letter}(\mathit{letter} \,|\, \mathit{digit})^* \\ \mathit{letter} &\rightarrow \mathrm{a} \,|\, \mathrm{b} \,|\, \mathrm{c} \,|\, \ldots \,|\, z \\ \mathit{digit} &\rightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9 \end{aligned}$$

"*start*", "*letter*", "*digit*" are part of a distinct set of identifiers called *non-terminals*, often denoted abstractly by letters P, Q, R, S, ... with S usually the Starting symbol

Regular grammars: *no recursion* (only use Kleene star)

Context-free grammars: *recursion is allowed*

Context-Free Grammars (CFG) and Derivations

Example CFG:

$$S \;\rightarrow\; \varepsilon \mid \mathrm{a}\,S\,\mathrm{b}$$

Semantics of CFGs given by *rewriting derivations*

Example derivation:

$$S \;\Rightarrow\; \mathrm{a}\,S\,\mathrm{b} \;\Rightarrow\; \mathrm{a\,a}\,S\,\mathrm{b\,b} \;\Rightarrow\; \mathrm{a\,a\,a}\,S\,\mathrm{b\,b\,b} \;\Rightarrow\; \mathrm{a\,a\,a}\,\varepsilon\,\mathrm{b\,b\,b} \;=\; \mathrm{a\,a\,a\,b\,b\,b}$$

Recap: Parse Trees And Abstract Syntax Trees

***Parse* trees** uniquely specify how an input was recognized by the grammar.

Parse trees *contain all information* needed to reconstruct the input.

Recap: Parse Trees And Abstract Syntax Trees

***Parse* trees** uniquely specify how an input was recognized by the grammar.

Parse trees *contain all information* needed to reconstruct the input.

Abstract syntax *trees* (ASTs) omit *syntactic details* not relevant to program semantics

AST can be pretty-printed,

but the result may not correspond to the specific input (loss of information)

Example: Expressions

An expression grammar:

$$
\begin{aligned}
\mathit{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid \text{`('}\ \mathit{expr}\ \text{`)'} \\
\mathit{op} &\rightarrow + \mid *
\end{aligned}
$$

A possible AST for it:

```
enum Expr:
  case IntLit(n: Int)
  case Var(name: String)
  case Add(e1: Expr, e2: Expr)
  case Mult(e1: Expr, e2: Expr)
```

Notice: no parenthesis case; no “op”

Recap: Ambiguities

Some grammars are ambiguous.

$$
\begin{aligned}
\mathit{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid \text{`('}\,\mathit{expr}\,\text{`)'} \\
\mathit{op} &\rightarrow + \mid *
\end{aligned}
$$

How to parse these?

- "$x * 42 + y$"
- "$x + 42 + y$"

Removing ambiguities requires transforming the grammar.

Generalities on Grammars

Chomskys Classification of Grammars (1959)

- Type 0, *unrestricted*: arbitrary string rewrite rules

  Equivalent to Turing machines!

  $eXb \rightarrow eXeX \rightarrow Y$

Chomskys Classification of Grammars (1959)

- Type 0, *unrestricted*: arbitrary string rewrite rules

  Equivalent to Turing machines!

  $eXb \rightarrow eXeX \rightarrow Y$

- Type 1, *context sensitive*: RHS always larger

  $O(n)$-space Turing machines

  $aXb \rightarrow acXb$

Chomskys Classification of Grammars (1959)

- Type 0, *unrestricted*: arbitrary string rewrite rules

  Equivalent to Turing machines!

  $eXb \longrightarrow eXeX \longrightarrow Y$

- Type 1, *context sensitive*: RHS always larger

  $O(n)$-space Turing machines

  $aXb \longrightarrow acXb$

- Type 2, *context free*: one LHS nonterminal

  $X \longrightarrow acXb$

Chomskys Classification of Grammars (1959)

- Type 0, *unrestricted*: arbitrary string rewrite rules

  Equivalent to Turing machines!

  $eXb \rightarrow eXeX \rightarrow Y$

- Type 1, *context sensitive*: RHS always larger

  $O(n)$-space Turing machines

  $aXb \rightarrow acXb$

- Type 2, *context free*: one LHS nonterminal

  $X \rightarrow acXb$

- Type 3, *regular*: no recursion, just Kleene star

  $X \rightarrow acY^*b$

Chomskys Classification of Grammars (1959)

- Type 0, *unrestricted*: arbitrary string rewrite rules

  Equivalent to Turing machines!

  $$eXb \rightarrow eXeX \rightarrow Y$$

- Type 1, *context sensitive*: RHS always larger

  $O(n)$-space Turing machines

  $$aXb \rightarrow acXb$$

- Type 2, *context free*: one LHS nonterminal

  $$X \rightarrow acXb$$

- Type 3, *regular*: no recursion, just Kleene star

  $$X \rightarrow acY^*b$$

Type 3 $\subset$ Type 2 $\subset$ Type 1 $\subset$ Type 0

Note on Parsing General Grammars

Decidable even for type 1 grammars (by eliminating epsilons – Chomsky, 1959)

Note on Parsing General Grammars

Decidable even for type 1 grammars (by eliminating epsilons – Chomsky, 1959)

Simple algorithm for CFGs: "CYK" (CockeYoungerKasami), takes $O(n^3)$ time

Better complexity possible (L. G. Valiant, 1975) – reduce to matrix multiplication

- n^k for k between 2 and 3

More practical algorithms known (J. Earley, 1968)

- Can work in quadratic or linear time for some well-behaved grammars

Note on Parsing General Grammars

Decidable even for type 1 grammars (by eliminating epsilons – Chomsky, 1959)

Simple algorithm for CFGs: "CYK" (CockeYoungerKasami), takes $O(n^3)$ time

Better complexity possible (L. G. Valiant, 1975) – reduce to matrix multiplication

n^k for k between 2 and 3

More practical algorithms known (J. Earley, 1968)

Can work in quadratic or linear time for some well-behaved grammars

However, parsing general grammars has ***little interest***

for *computer* language processing, and in particular for *compiler design*.

Most *reasonable* programming languages have *reasonable* grammars!

Recursive Descent
LL(1) Parsing

- useful parsing technique
- to make it work, we might need to transform the grammar

A Rule of While Language Syntax

// Where things work very nicely for recursive descent!

statmt ::=

```
    println ( stringConst , ident )
  | ident = expr
  | if ( expr ) statmt (else statmt)?
  | while ( expr ) statmt
  | { statmt* }
```

Parser for the statmt (rule -> code)

```
def skip(t : Token) = if (lexer.token == t) lexer.next
  else error("Expected"+ t)
def statmt = {
  if (lexer.token == Println) { lexer.next;
    skip(openParen); skip(stringConst); skip(comma);
    skip(identifier); skip(closedParen)
  } else if (lexer.token == Ident) { lexer.next;
    skip(equality); expr
  } else if (lexer.token == ifKeyword) { lexer.next;
    skip(openParen); expr; skip(closedParen); statmt;
    if (lexer.token == elseKeyword) { lexer.next; statmt }
  // | while ( expr ) statmt
```

Continuing Parser for the Rule

// | while (expr) statmt

} **else if** (lexer.token == whileKeyword) { lexer.next;
skip(openParen); expr; skip(closedParen); statmt

// | { statmt }*

} **else if** (lexer.token == openBrace) { lexer.next;
while (**isFirstOfStatmt**) { statmt }
skip(closedBrace)

} **else** { error("Unknown statement, found token " +
lexer.token) }

How to construct **if** conditions?

statmt ::= println (stringConst , ident)
| if (expr) statmt (else statmt)$^?$
| while (expr) statmt

- Look what each alternative starts with to decide what to parse
- Here: we have terminals at the beginning of each alternative
- More generally, we have 'first' computation, as for regular expressions
- Consider a grammar G and non-terminal N

L_G(N) = { set of strings that N can derive }

e.g. L(statmt) – all statements of while language

first(N) = { a | aw in L_G(N), a – terminal, w – string of terminals}

first(statmt) = { **println**, **ident**, **if**, **while**, **{** }

first(**while** (expr) statmt) = { **while** } - we will give an algorithm

Formalizing and Automating Recursive Descent: LL(1) Parsers

Task: Rewrite Grammar to make it suitable for recursive descent parser

- Assume the priorities of operators as in Java

```
expr ::= expr (+|-|*|/) expr
       | name | `(' expr `)'
name ::= ident
```

Grammar vs Recursive Descent Parser

```
expr ::= term termList
termList ::= + term termList
          |  - term termList
          | ε
term ::= factor factorList
factorList ::= * factor factorList
             | / factor factorList
             | ε
factor ::= name | ( expr )
name ::= ident
```

```
def expr = { term; termList }
def termList =
  if (token==PLUS) {
    skip(PLUS); term; termList
  } else if (token==MINUS)
    skip(MINUS); term; termList
  }
def term = { factor; factorList }
...
def factor =
  if (token==IDENT) name
  else if (token==OPAR) {
   skip(OPAR); expr; skip(CPAR)
  } else error("expected ident or )")
```

Note that the abstract trees we would create in this example do not strictly follow parse trees.

Rough General Idea

$$
\begin{aligned}
A ::= \;& B_1 \ldots B_p \\
\mid\; & C_1 \ldots C_q \\
\mid\; & D_1 \ldots D_r
\end{aligned}
$$

$\Rightarrow$

def A =
  if (token $\in$ T1) {
    B_1 ... B_p
  else if (token $\in$ T2) {
    C_1 ... C_q
  } **else if** (token $\in$ T3) {
    D_1 ... D_r
  } **else** error("expected T1,T2,T3")

where:

T1 = **first**(B_1 ... B_p)

T2 = **first**(C_1 ... C_q)

T3 = **first**(D_1 ... D_r)

first(B_1 ... B_p) = $\{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

T1, T2, T3 should be **disjoint** sets of tokens.

Computing **first** in the example

expr ::= term termList
termList ::= **+** term termList
  | **-** term termList
  | ε
term ::= factor factorList
factorList ::= * factor factorList
  | / factor factorList
  | ε
factor ::= name | **(** expr **)**
name ::= **ident**

first(name) = {**ident**}
first(**(** expr **)**) = { **(** }
first(factor) = first(name)
  U first(**(** expr **)**)
  = {**ident**} U{ **(** }
  = {**ident**, **(** }
first(* factor factorList) = { * }
first(/ factor factorList) = { / }
first(factorList) = { *, / }
first(term) = first(factor) = {**ident**, **(** }
first(termList) = { **+** , **-** }
first(expr) = first(term) = {**ident**, **(** }

Algorithm for **first**: Goal

Given an arbitrary context-free grammar with a set of rules of the form $X ::= Y_1 \ldots Y_n$ compute first for each right-hand side and for each symbol.

How to handle

- alternatives for one non-terminal
- sequences of symbols
- nullable non-terminals
- recursion

Rules with Multiple Alternatives

$$\boxed{\begin{aligned} A ::= \; & B_1 \dots B_p \\ | \; & C_1 \dots C_q \\ | \; & D_1 \dots D_r \end{aligned}} \Rightarrow \boxed{\begin{aligned} first(A) = \; & first(B_1 \dots B_p) \\ & \cup\, first(C_1 \dots C_q) \\ & \cup\, first(D_1 \dots D_r) \end{aligned}}$$

Sequences

$\boxed{first(B_1 \dots B_p) = first(B_1)}$ if not nullable(B_1)

$\boxed{first(B_1 \dots B_p) = first(B_1) \cup \dots \cup first(B_k)}$

if nullable(B_1), ..., nullable(B_{k-1}) and

not nullable(B_k) or k=p

Abstracting into Constraints

recursive grammar: constraints over finite sets: expr' is first(expr)

recursive grammar	constraints
expr ::= term termList	expr' = term'
termList ::= **+** term termList	termList' = {**+**}
| **-** term termList	U {**-**}
| ε	
term ::= factor factorList	term' = factor'
factorList ::= * factor factorList	factorList' = { * }
| / factor factorList	U { / }
| ε	
factor ::= name | **(** expr **)**	factor' = name' U { **(** }
name ::= **ident**	name' = { **ident** }

nullable: termList, factorList

For this nice grammar, there is no recursion in constraints. Solve by substitution.

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

S' = X' U Y'
X' =

terminals: **a**,**b**
non-terminals: S, X, Y, Z

reachable (from S):
productive:
nullable:

First sets of terminals:
S', X', Y', Z' ⊆ {a,b}

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

$\Rightarrow$

S' = X' U Y'
X' = {b} U S'
Y' = Z' U **X'** U Y'
Z' = {a}

terminals: **a**,**b**
non-terminals: S, X, Y, Z

reachable (from S): S, X, Y, Z
productive: X, Z, S, Y
nullable: Z

These constraints are recursive.
How to solve them?
$S', X', Y', Z' \subseteq \{a,b\}$
How many candidate solutions
- in this case?
- for k tokens, n nonterminals?

Iterative Solution of **first** Constraints

	S'	X'	Y'	Z'
1.	{}	{}	{}	{}
2.	{}	{b}	{b}	{a}
3.	{b}	{b}	{a,b}	{a}
4.	{a,b}	{a,b}	{a,b}	{a}
5.	{a,b}	{a,b}	{a,b}	{a}

S' = X' U Y'
X' = {b} U S'
Y' = Z' U **X'** U Y'
Z' = {a}

- Start from all sets empty.
- Evaluate right-hand side and assign it to left-hand side.
- Repeat until it stabilizes.

Sets grow in each step

- initially they are empty, so they can only grow
- if sets grow, the RHS grows (U is monotonic), and so does LHS
- they cannot grow forever: in the worst case contain all tokens

Constraints for Computing Nullable

- Non-terminal is nullable if it can derive ε

S ::= X | Y X ::= **b** | S Y Y ::= Z X **b** | Y **b** Z ::= ε | **a**	⇒	S' = X' | Y' X' = 0 | (S' & Y') Y' = (Z' & X' & 0) | (Y' & 0) Z' = 1 | 0

S', X', Y', Z' ∈ {0,1}

0 - not nullable

1 - nullable

| - disjunction

& - conjunction

	S'	X'	Y'	Z'
1.	0	0	0	0
2.	0	0	0	1
3.	0	0	0	1

again monotonically growing

Computing first and nullable

- Given any grammar we can compute
  - for each non-terminal X whether nullable(X)
  - using this, the set first(X) for each non-terminal X
- General approach:
  - generate constraints over finite domains, following the structure of each rule
  - solve the constraints iteratively
    - start from least elements
    - keep evaluating RHS and re-assigning the value to LHS
    - stop when there is no more change

Summary: Algorithm for **nullable**

```
nullable = {}
changed = true
while (changed) {
 changed = false
 for each non-terminal X
  if ((X is not nullable) and
      (grammar contains rule    X ::= ε | ...    )
         or  (grammar contains rule    X ::= Y1 ... Yn | ...
      where {Y1,...,Yn} ⊆ nullable)
  then {
     nullable = nullable U {X}
     changed = true
   }
}
```

Summary: Algorithm for **first**

for each nonterminal X: first(X)={}

for each terminal t: first(t)={t}

repeat

  for each grammar rule X ::= Y(1) … Y(k)

  for i = 1 to k

    if i=1 or {Y(1),…,Y(i-1)} ⊆ nullable **then**

      first(X) = first(X) U first(Y(i))

until none of first(…) changed in last iteration

Follow sets. LL(1) Parsing Table

Exercise Introducing Follow Sets

Compute nullable, first for this grammar:

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Describe a parser for this grammar and explain how it behaves on this input:

beginof myPrettyCode

x = u;

y = v;

myPrettyCode **ends**

How does a recursive descent parser look like?

```
def stmtList =
  if (???) {}
  else { stmt; stmtList }
def stmt =
  if (lex.token == ID) assign
  else if (lex.token == beginof) block
  else error("Syntax error: expected ID or beginonf")
...
def block =
  { skip(beginof); skip(ID); stmtList; skip(ID); skip(ends) }
```

what should the condition be?

Problem Identified

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Problem parsing stmtList:

- **ID** could start alternative stmt stmtList
- **ID** could **follow** stmt, so we may wish to parse **ε** that is, do nothing and return

- For nullable non-terminals, we must also compute what **follows** them

LL(1) Grammar - good for building recursive descent parsers

- Grammar is LL(1) if for each nonterminal X
  - first sets of different alternatives of X are disjoint
  - if nullable(X), first(X) must be disjoint from follow(X) and only one alternative of X may be nullable
- For each LL(1) grammar we can build recursive-descent parser
- Each LL(1) grammar is unambiguous
- If a grammar is not LL(1), we can sometimes transform it into equivalent LL(1) grammar

Computing if a token can **follow**

$\mathbf{first}(B_1 \ldots B_p) = \{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

$\mathbf{follow}(X) = \{a \in \Sigma \mid S \Rightarrow \ldots \Rightarrow \ldots Xa \ldots\}$

There exists a derivation from the start symbol that produces a sequence of terminals and nonterminals of the form ...Xa...
(the token a follows the non-terminal X)

Rule for Computing Follow

Given $X ::= YZ$ (for reachable X)

then **first**$(Z) \subseteq$ **follow**(Y)

and **follow**$(X) \subseteq$ **follow**(Z)

now take care of nullable ones as well:

For each rule $X ::= Y_1 \ldots Y_p \ldots Y_q \ldots Y_r$

follow(Y_p) should contain:

- **first**$(Y_{p+1}Y_{p+2}\ldots Y_r)$
- also **follow**(X) if **nullable**$(Y_{p+1}Y_{p+2}Y_r)$

Compute nullable, first, follow

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Is this grammar LL(1)?

Conclusion of the Solution

The grammar is not LL(1) because we have

- nullable(stmtList)
- first(stmt) ∩ follow(stmtList) = {**ID**}

- If a recursive-descent parser sees **ID**, it does not know if it should
  - finish parsing stmtList or
  - parse another stmt

Table for LL(1) Parser: Example

S ::= B **EOF**

(1)

B ::= ε | B **(**B**)**

(1) **(2)**

empty entry:
when parsing S,
if we see) ,
report error

nullable: B

first(S) = { **(**, **EOF** }

follow(S) = {}

first(B) = { **(** }

follow(B) = { **)**, **(**, **EOF** }

Parsing table:

	EOF	(	)
S	{1}	{1}	{}
B	{1}	{1,2}	{1}

parse conflict - choice ambiguity:
grammar not LL(1)

1 is in entry because **(** is in follow(B)

2 is in entry because **(** is in first(B(B))

Table for LL(1) Parsing

Tells which alternative to take, given current token:

choice : Nonterminal x Token -> Set[Int]

$A ::= (1)\ B_1 \ldots B_p$
$\quad | \ (2)\ C_1 \ldots C_q$
$\quad | \ (3)\ D_1 \ldots D_r$

if $t \in first(C_1 \ldots C_q)$ add 2 to choice(A,t)

if $t \in follow(A)$ add K to choice(A,t) where K is nullable

For example, when parsing A and seeing token t

choice(A,t) = {2} means: parse alternative 2 $(C_1 \ldots C_q)$

choice(A,t) = {3} means: parse alternative 3 $(D_1 \ldots D_r)$

choice(A,t) = {} means: report syntax error

choice(A,t) = {2,3} : not LL(1) grammar

General Idea when parsing nullable(A)

$$
\begin{aligned}
A ::= \; & B_1 \ldots B_p \\
| \; & C_1 \ldots C_q \\
| \; & D_1 \ldots D_r
\end{aligned}
$$

$\Rightarrow$

def A =
  if (token $\in$ T1) {
    $B_1 \ldots B_p$
  else if (token $\in$ **(**T2 U T_F**)**) {
    $C_1 \ldots C_q$
  } **else if** (token $\in$ T3) {
    $D_1 \ldots D_r$
  } // no else error, just return

where:

T1 = **first**($B_1 \ldots B_p$)
T2 = **first**($C_1 \ldots C_q$)
T3 = **first**($D_1 \ldots D_r$)
T_F = **follow**(A)

Only one of the alternatives can be nullable (here: 2nd)
T1, T2, T3, T_F should be pairwise **disjoint** sets of tokens.

Concrete Parser Implementation

Concrete Parser Implementation

In practice, also want to ***produce*** *abstract syntax trees*,

not just recognize languages!

⇒ Make our recursive-descent methods return AST instances

How to concisely deal with parser state?

Example Language

Consider the following definitions:

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

// "A + B * C"  ⇒  Ident("A"),Plus,Ident("B"),Times,Ident("C")

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

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

Mutable Parser Architecture

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

    // Parser state manipulation:
    var cur: Option[Token] = ite.nextOption
    def consume: Unit =
      cur = ite.nextOption

    // define parser here:
    def expr = ...

object Parser:
    def parse(ts: Iterable[Token]): Expr =
      val p = Parser(ts.iterator)
      val res = p.expr // entry point
      if p.cur.nonEmpty then fail("input not fully consumed")
      res
```

Parsing Atomic Expressions

```
// Helper method:
def skip(tk: Token): Unit =
  if cur != Some(tk)
      then fail("expected " + tk + ", found " + cur)
  consume

// Unambiguous "atomic" expressions:
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)
```

Implementing Precedence and Associativity Right

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

then *reassociate* these lists correctly.

```
def atom = ... // as before
def multipliedAtoms: List[Expr] = ??? // parse:  * atom * atom * ...
def product: Expr = ???                // parse:  atom * atom * ...
def addedProducts: List[Expr] = ???    // parse:  + prod + prod + ...
def expr: Expr = ???                   // parse:  prod + prod + ...
```

Implementing Precedence and Associativity Right

```
def expr: Expr =
  val p = product; val ps = addedProducts
  ps.foldLeft(p)((l, r) ⇒ Add(l, r))

def addedProducts: List[Expr] = cur match
  case Some(Plus) ⇒
    consume; product :: addedProducts
  case _ ⇒ Nil

def product: Expr =
  val a = atom; val as = multipliedAtoms
  as.foldLeft(a)((l, r) ⇒ Mult(l, r))

def multipliedAtoms: List[Expr] = cur match
  case Some(Times) ⇒
    consume; atom :: multipliedAtoms
  case _ ⇒ Nil
```

Note: Debugging Parsers in Scala

Very simple yet effective ways of debugging Scala implementations:

Note: Debugging Parsers in Scala

Very simple yet effective ways of debugging Scala implementations:

- Use the pprint library to display readable trees.

```
pprint.log(ExprParser.parse(ts))
```

```
Test2.scala:49 ExprParser.parse(ts): Infix(
  lhs = Infix(
    lhs = Infix(lhs = Var(name = "A"), op = "*", rhs = Var(name = "B")),
    op = "-",
    rhs = Var(name = "C")
  ),
  op = "/",
  rhs = Var(name = "D")
)
```

Note: Debugging Parsers in Scala

Very simple yet effective ways of debugging Scala implementations:

- Use the pprint library to display readable trees.

```
pprint.log(ExprParser.parse(ts))
```

```
Test2.scala:49 ExprParser.parse(ts): Infix(
  lhs = Infix(
    lhs = Infix(lhs = Var(name = "A"), op = "*", rhs = Var(name = "B")),
    op = "-",
    rhs = Var(name = "C")
  ),
  op = "/",
  rhs = Var(name = "D")
)
```

- Use the sourcecode library to display line numbers or definition names.

Note: Debugging Parsers in Scala

- Use the sourcecode library to display line numbers or definition names.

Instrumenting helper methods:

```
class TypeParser(ite: Iterator[Token], debug: Boolean):

  var _cur = ite.nextOption

  def cur(using n: sourcecode.Name) =
    if debug then println(s"⇒ ${n.value}\tinspects ${_cur}")
    _cur

  def consume(using n: sourcecode.Name) =
    if debug then println(s"⇒ ${n.value}\t  consumes ${_cur}")
    _cur = ite.nextOption

 ...
```

Note: Debugging Parsers in Scala

- Use the sourcecode library to display line numbers or definition names.

Creates a *trace* of the parser execution without any modification to parser definitions!

```
⇒ expr       inspects Some(OpenParen)
⇒ expr         consumes Some(OpenParen)
⇒ expr       inspects Some(OpenParen)
⇒ expr         consumes Some(OpenParen)
⇒ expr       inspects Some(Ident(A))
⇒ expr         consumes Some(Ident(A))
⇒ exprCont   inspects Some(Oper(*))
⇒ exprCont     consumes Some(Oper(*))
⇒ expr       inspects Some(Ident(B))
⇒ expr         consumes Some(Ident(B))
⇒ exprCont   inspects Some(Oper(-))
⇒ exprCont   inspects Some(Oper(-))
⇒ exprCont     consumes Some(Oper(-))
...
```

Note: Removing Mutation

Mutation used in these slides for conciseness.

Note: Removing Mutation

Mutation used in these slides for conciseness.

Refer to Tutorial 4 solutions for a *purely functional* parser (working on *lists* of tokens)

```
def ty(ts: List[Token]): (Type, List[Token]) =
  val (us, rest) = unions(ts)
  rest match
    case Arrow :: rest ⇒
      val (t, rest2) = ty(rest); (Infix(us, Fun, t), rest2)
    case _ ⇒ (us, rest)
```

Note: Removing Mutation

Mutation used in these slides for conciseness.

Refer to Tutorial 4 solutions for a *purely functional* parser (working on *lists* of tokens)

```
def ty(ts: List[Token]): (Type, List[Token]) =
  val (us, rest) = unions(ts)
  rest match
    case Arrow :: rest ⇒
      val (t, rest2) = ty(rest); (Infix(us, Fun, t), rest2)
    case _ ⇒ (us, rest)
```

To make such parser *streaming*, simply use a Scala *LazyList* instead of a *List*

Note: Removing Mutation

Mutation used in these slides for conciseness.

Refer to Tutorial 4 solutions for a *purely functional* parser (working on *lists* of tokens)

```
def ty(ts: List[Token]): (Type, List[Token]) =
  val (us, rest) = unions(ts)
  rest match
    case Arrow :: rest ⇒
      val (t, rest2) = ty(rest); (Infix(us, Fun, t), rest2)
    case _ ⇒ (us, rest)
```

To make such parser *streaming*, simply use a Scala *LazyList* instead of a *List*

Lookahead simpler to implement:

```
case Ident(nme) :: OpenBracket :: rest ⇒ ...
```