Exercise 1

Consider a language with the following tokens and token classes:

ID ::= letter (letter|digit)*

LT ::= "<"

GT ::= ">"

shiftL ::= "<<"

shiftR ::= ">>"

dot ::= "."

LP ::= "("

RP ::= ")"

Give a sequence of tokens for the following character sequence, applying the longest match rule:

(List<List<Int>>)(myL).headhead

Note that the input sequence contains no space character

Exercise 2

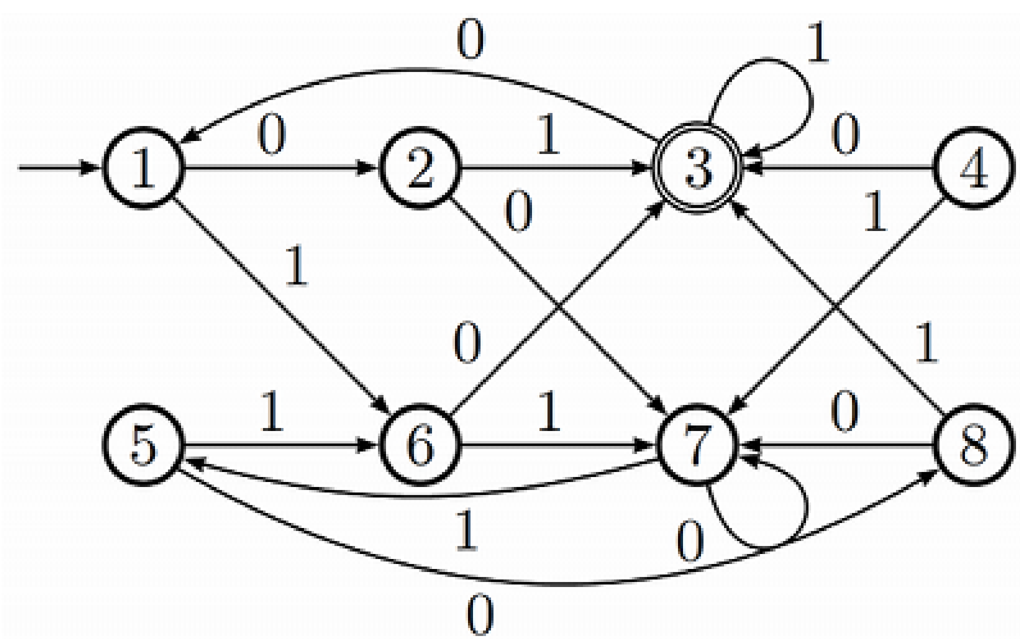
Find a regular expression that generates all alternating sequences of 0 and 1 with arbitrary length **(including lengths zero, one, two, ...)**. For example, the alternating sequences of length one are 0 and 1, length two are 01 and 10, length three are 010 and 101. Note that no two adjacent character can be the same in an alternating sequence.

Exercise 3

Construct a DFA (deterministic finite-state automaton) for the language L of *well-nested* parenthesis of nesting depth at most 3. For example,
ε, ()(), (()(())) and (()())()() should be in L,
but not (((()))) nor (()(()(()))), nor ())) .

Exercise 4

- Find two equivalent states in the automaton, and merge them to produce a smaller automaton that recognizes the same language. Repeat until there are no longer equivalent states.
- Recall that the general algorithm for minimizing finite automata works in reverse. First, find all pairs of inequivalent states. States X, Y are inequivalent if X is final and Y is not, or (by iteration) if and and X’ and Y’ are inequivalent. After this iteration ceases to find new pairs of inequivalent states, then X, Y are equivalent, if they are not inequivalent.



Exercise 5

Let *tail* be a function that returns all the symbols of a string except the last one. For example

tail(mama)=mam

tail is undefined for an empty string. If $L_1 \subseteq A^*$, then $TAIL(L_1)$ applies the function to all non-empty words in L_1, ignoring ε if it is in L_1: $TAIL(L_1) = \{ v \in A^* \mid \exists\, a \in A.\ va \in L_1\}$

TAIL({aba,aaaa,bb, ε}) = {ab,aaa,b}

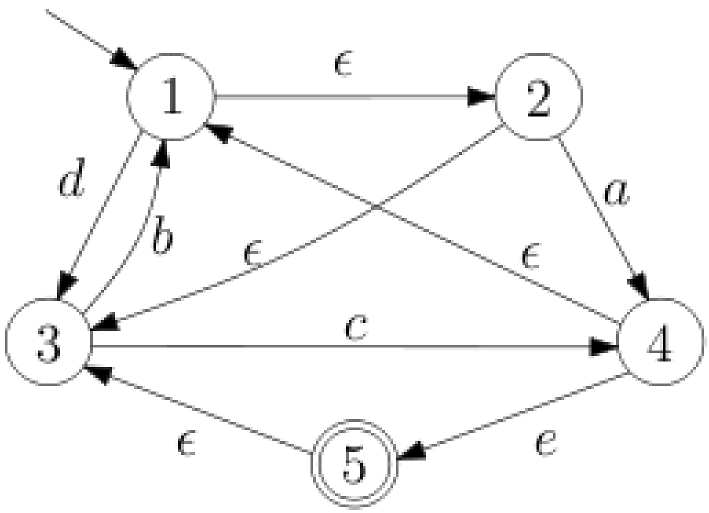
L(r) denotes the language of a regular expression r. Then TAIL(L(abba|ba*|ab*)) = L(ba*|ab*|ε)

Tasks:

- Prove that if language L_1 is regular, then so is $TAIL(L_1)$
- Give an algorithm that, given a regular expression r for L_1, computes a regular expression rtail(r) for language $TAIL(L_1)$

Exercise 6. Given NFA A, find first(L(A))

- Compute the set of first symbols of words accepted by the following non-deterministic finite state machine with epsilon transitions:



- Describe an algorithm that solves this problem given a given NFA

More Questions

- Find automaton or regular expression for:
  - Any sequence of open and closed parentheses of even length?
  - as many digits before as after decimal point?
  - Sequence of balanced parentheses
    ((()) ()) - balanced
    ( )) (( ) - not balanced
  - Comment as a sequence of space,LF,TAB, and comments from // until LF
  - Nested comments like /* ... /* */ ... */

Automaton that Claims to Recognize $\{ a^n b^n \mid n >= 0 \}$

Make the automaton deterministic

Let the resulting DFA have K states, $|Q|=K$

Feed it a, aa, aaa, …. Let q_i be state after reading a^i

$q_0, q_1, q_2, \dots, q_K$

This sequence has length K+1 -> a state must repeat

$q_i = q_{i+p}$ $\quad\quad\quad\quad$ $p > 0$

Then the automaton should accept $a^{i+p}b^{i+p}$.

But then it must also accept

$$a^i \, b^{i+p}$$

because it is in state after reading a^i as after a^{i+p}.

So it does not accept the given language.

Limitations of Regular Languages

- Every automaton can be made deterministic
- Automaton has finite memory, cannot count
- Deterministic automaton from a given state behaves always the same
- If a string is too long, deterministic automaton will repeat its behavior

Pumping Lemma

If L is a regular language, then there exists a positive integer p (the pumping length) such that every string $s \in$ L for which $|s| \geq p$, can be partitioned into three pieces, $s = x\ y\ z$, such that

- $|y| > 0$
- $|xy| \leq p$
- $\forall i \geq 0.\ xy^i z \in$ L

Let's try again: $\{\ a^n b^n\ \mid n >= 0\ \}$

Automata are Limited

Let us use **grammars**!

Context Free Grammar for $a^n b^n$

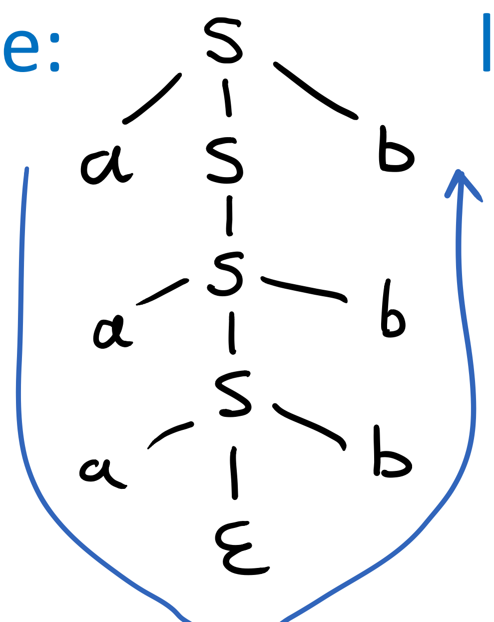
$S ::= \varepsilon$

$S ::= a\ S\ b$

Example of a derivation

$S \Rightarrow aSb \Rightarrow a\ aSb\ b \Rightarrow aa\ aSb\ bb \Rightarrow aaabbb$

Derivation tree:



leaves give us result

aaabbb

Context-Free Grammars

G = (A, N, S, R)

- A - terminals (alphabet for generated words $w \in A^*$)
- N - non-terminals – symbols with recursive definitions
- Grammar rules in R are pairs, written ss

  n ::= v where

  $n \in N$ is a non-terminal

  $v \in (A \cup N)^*$ - sequence of terminals and non-terminals

A derivation in G starts from the starting symbol S

- Each step replaces a non-terminal with one of its right hand sides

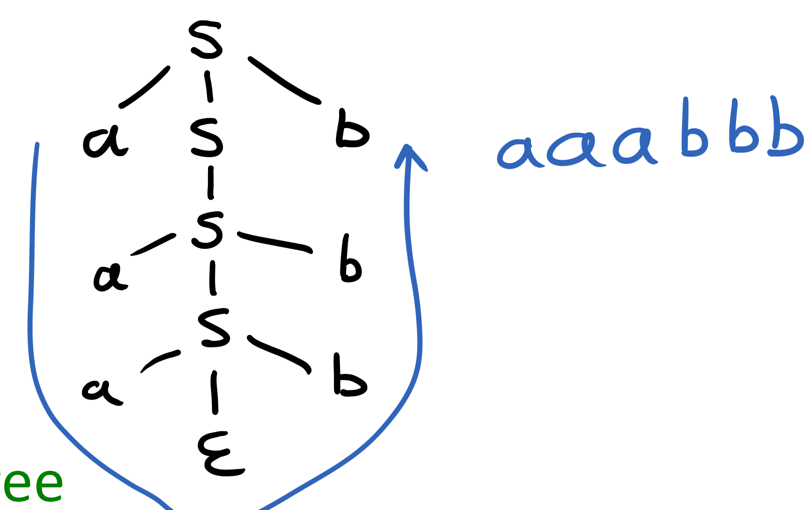
Example from before: G = ({a,b}, {S}, S, {S::= ε, S::=aSB})

Parse Tree

Given a grammar G = (A, N, S, R), t is a **parse tree** of G (isParseTree) if t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of A
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n whose children are labelled by $p_1 \ldots p_n$, we have a rule $(n ::= p_1 \ldots p_n) \in R$

Yield of a parse tree t is the unique word in A* obtained by reading the leaves of t from left to right



Language of a grammar G =
words of all yields of parse trees of G

L(G) = {yield(t) | isParseTree(G,t)}

isParseTree - **easy** to check

Harder: know if a word has a parse tree

Grammar Derivation

A **derivation** for G is any sequence of words $p_i \in (A \cup N)^*$,whose:

- first word is S
- each subsequent word is obtained from the previous one by replacing one of its letters by right-hand side of a rule in R :
  $p_i = unv$, $(n::=q) \in R$,
  $p_{i+1} = uqv$
- Last word has only letters from A

Each parse tree of a grammar has one or more derivations, which result in expanding tree gradually from S

- Different orders of expanding non-terminals may generate the same tree

Example: Parse Tree vs Derivation

Consider this grammar G = ({a,b}, {S,P,Q}, S, R) where R is:

S ::= PQ

P ::= a

P ::= aP

Q ::= aQb

Q ::= ε

Show a derivation tree for aaaabb

Show at least two derivations that correspond to that tree.

Balanced Parentheses Grammar

Consider the language L consisting of precisely those words consisting of parentheses “**(**“ and “**)**” that are balanced (each parenthesis has the matching one)

- Example sequence of parentheses

((()) ()) - balanced, belongs to the language

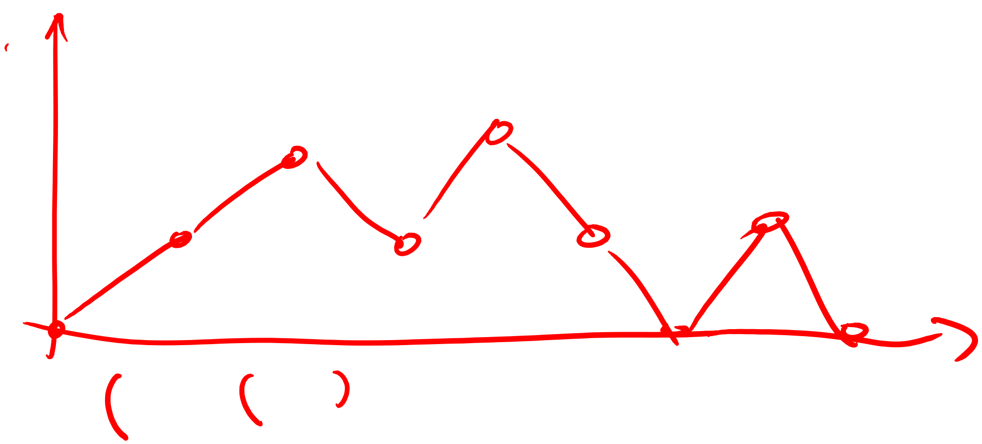
( )) (( ) - not balanced, does not belong

Exercise: give the grammar and example derivation for first language.

Balanced Parentheses Grammar

$$S ::= \ \mid\ \varepsilon\ \mid\ (S)S$$

$$S \to (S)S \to (\varepsilon)S \to ()S \to ()(S)S$$
$$\to ()()$$



(()())()

Proving Grammar Defines a Language

Grammar G: $S ::= \varepsilon$, $S ::=$ **(**S**)**S

defines language L(G)

Theorem: $L(G) = L_b$

where $L_b = \{ w \mid$ for every pair u,v of words such that $uv=w$, the number of **(** symbols in u is greater or equal than the number of **)** symbols in u . These numbers are equal in w $\}$

$L(G) \subseteq L_b$: If $w \in L(G)$, then it has a parse tree. We show $w \in L_b$ by induction on size of the parse tree deriving w using G.

If tree has one node, it is "", and "" $\in L_b$, so we are done.

Suppose property holds for trees up size n. Consider tree of size n. The root of the tree is given by rule **(**S**)**S . The derivation of sub-trees for the first and second S belong to L_b by induction hypothesis. The derived word w is of the form (p)q where p,q$\in L_b$. Let us check if (p)q $\in L_b$. Let (p)q = uv and count the number of **(** and **)** in u. If u then it satisfies the property. If it is shorter than |p|+1 then it has at least one more **(** than **)**. Otherwise it is of the form (p)q_1 where q_1 is a prefix of q. Because the parentheses balance out in p and thus in (p), the difference in the number of **(** and **)** is equal to the one in q_1 which is a prefix of q so it satisfies the property. Thus u satisfies the property as well.

$L_b \subseteq L(G)$: If $w \in L_b$, we need to show that it has a parse tree. We do so by induction on $|w|$. If w="" then it has a tree of size one (only root). Otherwise, suppose all words of length <n have parse tree using G. Let $w \in L_b$ and $|w|=n>0$. (Please refer to the figure counting the difference between the number of **(** and **)**. We split w in the following way: let p_1 be the shortest non-empty prefix of w such that the number of **(** equals to the number of **)**. Such prefix always exists and is non-empty, but could be equal to w itself. Note that it must be that p_1 = **(**p**)** for some p because p_1 is a prefix of a word in L_b , so the first symbol must be **(** and, because the final counts are equal, the last symbol must be **)**. Therefore, w = **(**p**)**q for some shorter words p,q. Because we chose p to be the shortest, prefixes of **(**p always have at least one more **(**. Therefore, prefixes of p always have at greater or equal number of **(**, so p is in L_b. Next, for prefixes of the form **(**p**)**v the difference between **(** and **)** equals this difference in v itself, since (p) is balanced. Thus, v has at least as many **(** as **)**. We have thus shown that w is of the form (p)q where p,q are in L_b. By IH p,q have parse trees, so there is parse tree for w.

Exercise: Grammar Equivalence

Show that each string that can be derived by grammar G_1

$$B ::= \varepsilon \mid \mathbf{(}\ B\ \mathbf{)} \mid B\ B$$

can also be derived by grammar G_2

$$B ::= \varepsilon \mid \mathbf{(}\ B\ \mathbf{)}\ B$$

and vice versa. In other words, $L(G_1) = L(G_2)$

Remark: there is no algorithm to check for equivalence of *arbitrary* grammars. We must be clever.

Regular Languages and Grammars

Exercise: give grammar describing the same language as this regular expression:

$$(a|b)\,(ab)^*b^*$$

Translating Regular Expression into a Grammar

- Suppose we first allow regular expression operators * and | within grammars
- Then R becomes simply

  S ::= R

- Then give rules to remove *, | by introducing new non-terminal symbols

Eliminating Additional Notation

- Alternatives

  s ::= P | Q   becomes   s ::= P

  s ::= Q

- Parenthesis notation – introduce symbol

  expr (&& | < | == | + | - | * | / | %) expr

- Kleene star

  { statmt* }

- Optional parts

  if (expr) statmt (else statmt)$^{?}$

Grammars for Natural Language

Statement = Sentence "."

Sentence ::= Simple | Belief

Simple ::= Person liking Person

liking ::= "likes" | "does" "not" "like"

Person ::= "Barack" | "Helga" | "John" | "Snoopy"

Belief ::= Person believing "that" Sentence but

believing ::= "believes" | "does" "not" "believe"

but ::= "" | "," "but" Sentence

→ can also be used to automatically generate essays

Exercise: draw the derivation tree for:

John does not believe that
    Barack believes that Helga likes Snoopy,
but Snoopy believes that Helga likes Barack.

While Language Syntax

This syntax is given by a context-free grammar:

program ::= statmt*

statmt ::= println(stringConst , ident)

| ident = expr

| **if** (expr) statmt (else statmt)$^?$

| **while** (expr) statmt

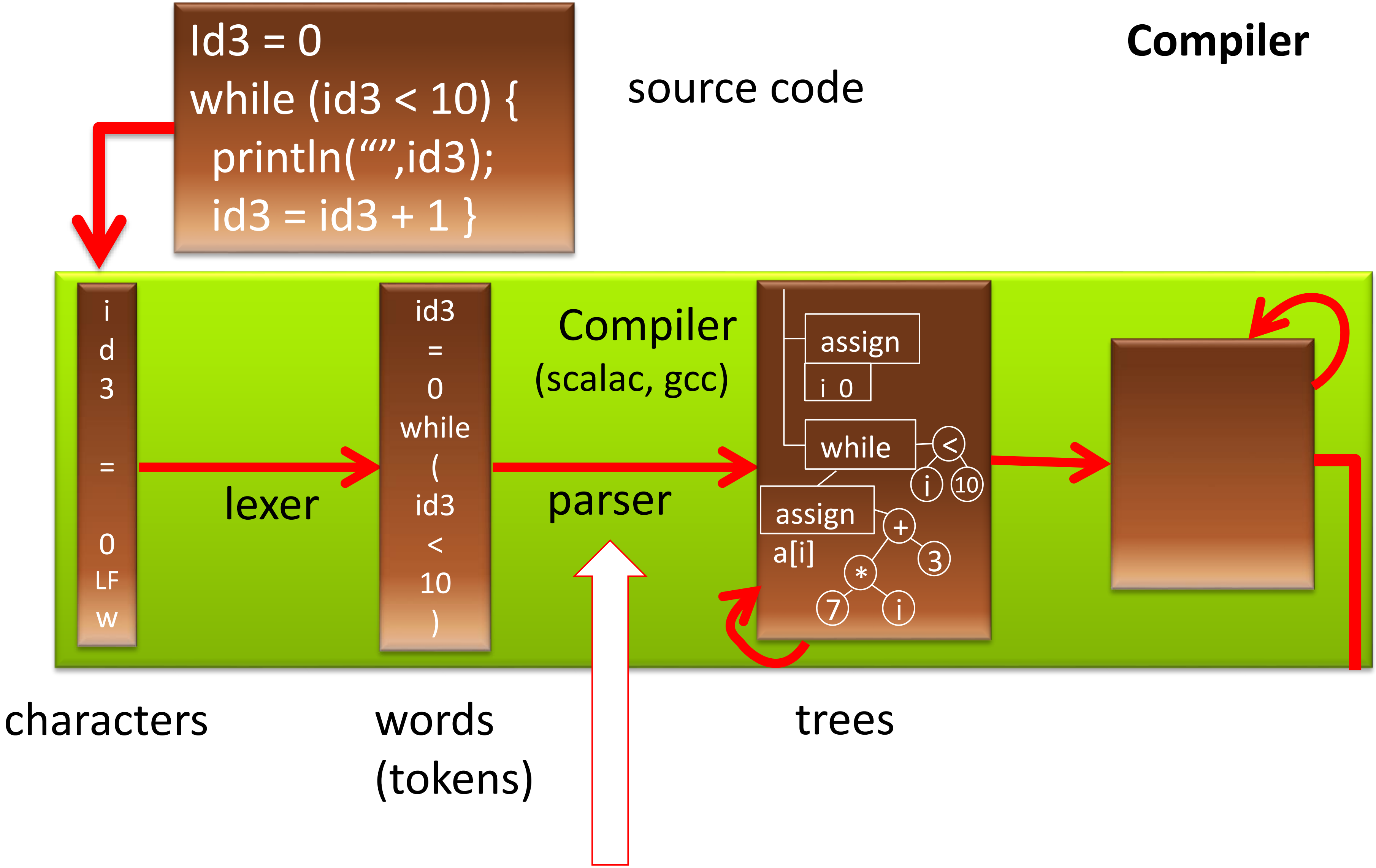
| { statmt* }

expr ::= intLiteral | ident

| expr (&& | < | == | + | - | * | / | %) expr

| ! expr | - expr

Compiler



Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees

Recursive Descent Parsing

Recursive Descent is Decent

descent = a movement downward

decent = adequate, good enough

Recursive descent is a decent parsing technique

- can be easily implemented manually based on the grammar (which may require transformation)
- efficient (linear) in the size of the token sequence

Correspondence between grammar and code

- concatenation → ;
- alternative (|) → if
- repetition (*) → while
- nonterminal → recursive procedure

A Rule of While Language Syntax

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)
// statmt ::=
def statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); skip(stringConst); skip(comma);
    skip(identifier); skip(closedParen)
  // | ident = expr
  } else if (lexer.token == Ident) { lexer.next;
    skip(equality); expr
  // | if ( expr ) statmt (else statmt)?
  } 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) }

First Symbols for Non-terminals

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

- Consider a grammar G and non-terminal N

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

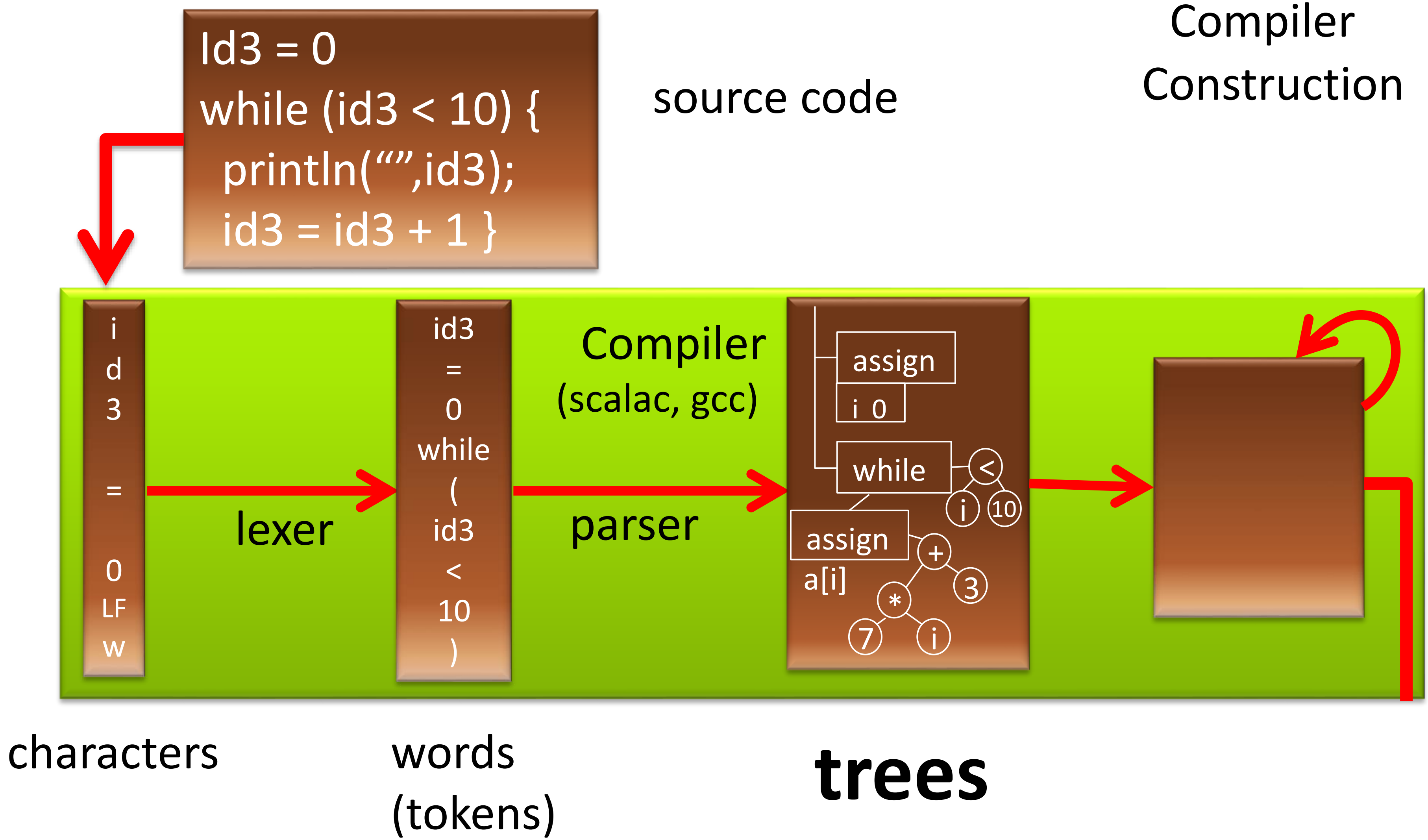
e.g. L(statmt) – all statements of while language

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

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

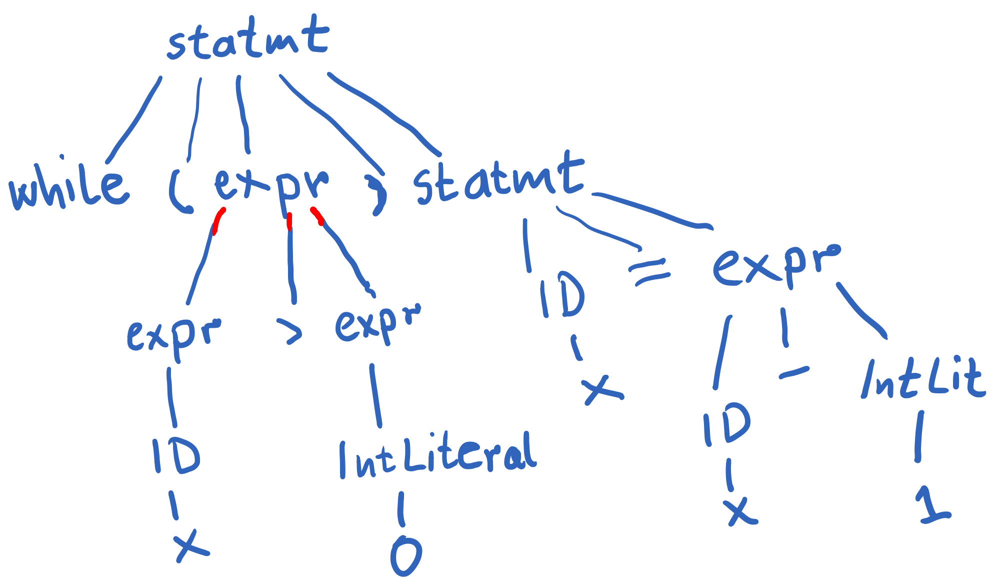
(we will see how to compute first in general)

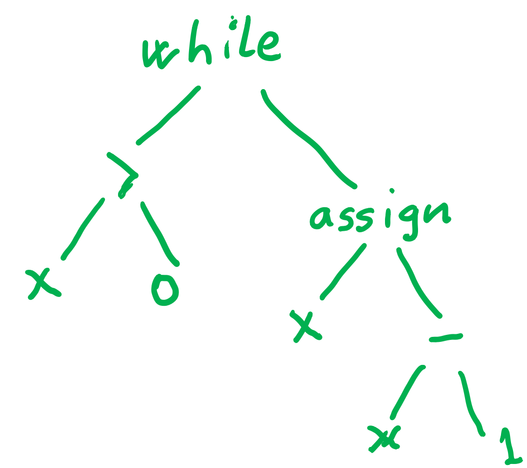
Compiler Construction



Parse Tree vs Abstract Syntax Tree (AST)

while (x > 0) x = x - 1





Pretty printer: takes abstract syntax tree (AST) and outputs the leaves of one possible (concrete) parse tree.

$$\text{parse(prettyPrint(ast))} \approx \text{ast}$$

Parse Tree vs Abstract Syntax Tree (AST)

- Each node in parse tree has children corresponding precisely to right-hand side of grammar rules
- Nodes in abstract syntax tree contain only useful information and usually omit e.g. the punctuation signs

Abstract Syntax Trees for Statements

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

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
                          falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Expr) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

Abstract Syntax Trees for Statements

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

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
              falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Statmt) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

Our Parser Produced Nothing ☹

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

Parser Returning a Tree ☺

```
def expect(t : Token) : Token = if (lexer.token == t) { lexer.next;t}
  else error("Expected"+ t)
// statmt ::=
def statmt : Statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); val s = getString(expect(stringConst));
    skip(comma);
    val id = getIdent(expect(identifier)); skip(closedParen)
    PrintlnS(s, id)
 // | ident = expr
  } else if (lexer.token.class == Ident) { val lhs = getIdent(lexer.token)
    lexer.next;
    skip(equality); val e = expr
    Assignment(lhs, e)
```

Constructing Tree for 'if'

```
def expr : Expr = { … }
// statmt ::=
def statmt : Statmt = {
  …
// if ( expr ) statmt (else statmt)?
// case class If(cond : Expr, trueBr: Statmt, falseBr: Option[Statmt])
  } else if (lexer.token == ifKeyword) { lexer.next;
     skip(openParen); val c = expr; skip(closedParen);
     val trueBr = statmt
     val elseBr = if (lexer.token == elseKeyword) {
        lexer.next; Some(statmt) } else None
     If(c, trueBr, elseBr)   // made a tree node ☺
  }
```

Task: Constructing Tree for ‘while’

```
def expr : Expr = { … }
// statmt ::=
def statmt : Statmt = {
  …
// while ( expr ) statmt
// case class While(cond : Expr, body : Expr) extends Statmt
} else if (lexer.token == WhileKeyword) {




} else
```

Here each alternative started with different token

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

What if this is not the case?

Left Factoring Example: Function Calls

statmt ::=

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

foo = 42 + x
foo (u , v)

code to parse the grammar:

```
} else if (lexer.token.class == Ident) {
    ???
}
```

Left Factoring Example: Function Calls

statmt ::=

println (stringConst , ident)

| ident assignmentOrCall

| if (expr) statmt (else statmt)?

| while (expr) statmt

| { statmt* }

assignmentOrCall ::= "=" expr | (expr (, expr)*)

code to parse the grammar:

```
} else if (lexer.token.class == Ident) {
    val id = getIdentifier(lexer.token); lexer.next
    assignmentOrCall(id)
}               // Factoring pulls common parts from alternatives
```

Beyond Statements: Parsing Expressions

While Language with Simple Expressions

statmt ::=

println (stringConst , ident)

| ident = expr

| if (expr) statmt (else statmt)$^?$

| while (expr) statmt

| { statmt* }

expr ::= intLiteral | ident

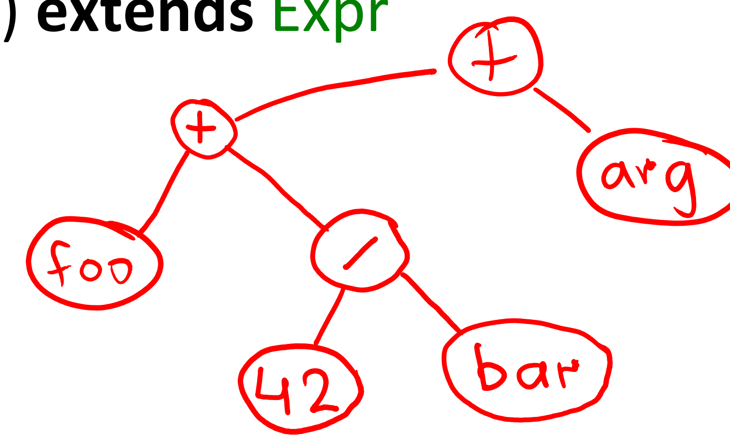
| expr (+ | /) expr

Abstract Syntax Trees for Expressions

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

```
abstract class Expr
case class IntLiteral(x : Int) extends Expr
case class Variable(id : Identifier) extends Expr
case class Plus(e1 : Expr, e2 : Expr) extends Expr
case class Divide(e1 : Expr, e2 : Expr) extends Expr
```

foo + 42 / bar + arg



Parser That Follows the Grammar?

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

input:

(foo + 42 / bar + arg)

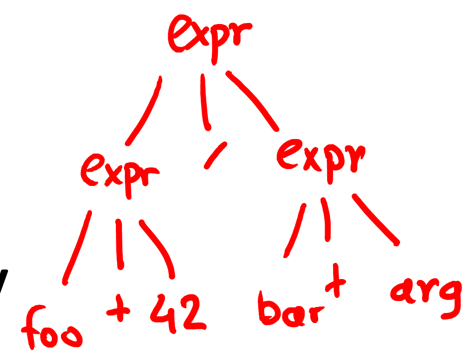
```
def expr : Expr = {
  if (??) IntLiteral(getInt(lexer.token))
  else if (??) Variable(getIdent(lexer.token))
  else if (??) {
    val e1 = expr; val op = lexer.token; val e2 = expr
    op match Plus {
      case PlusToken => Plus(e1, e2)
      case DividesToken => Divides(e1, e2)
    } }
```

When should parser enter the recursive case?!

Ambiguous Grammars

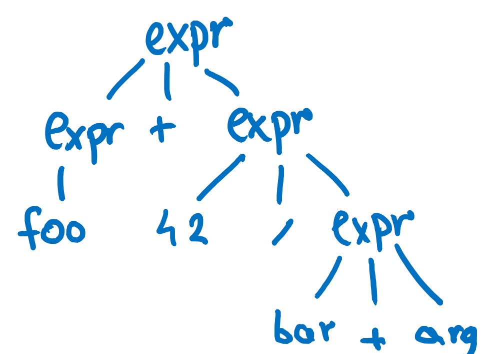
expr ::= intLiteral | ident
        | expr + expr | expr / expr
        | (expr)

foo + 42 / bar + arg



Each node in parse tree is given by one grammar alternative.

Ambiguous grammar: if some token sequence has multiple parse trees (then it is has multiple abstract trees).



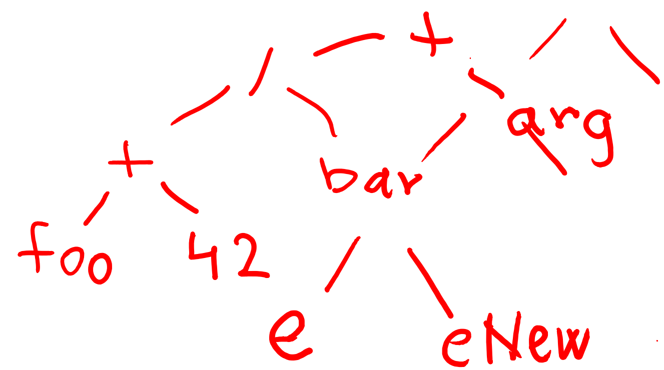
An attempt to rewrite the grammar

expr ::= simpleExpr ((+ | /) simpleExpr)*

simpleExpr ::= intLiteral | ident

```
def simpleExpr : Expr = { ... }
def expr : Expr = {
  var e = simpleExpr
  while (lexer.token == PlusToken ||
         lexer.token == DividesToken)) {
    val op = lexer.token
    val eNew = simpleExpr
    op match {
     case TokenPlus => { e = Plus(e, eNew) }
     case TokenDiv => { e = Divide(e, eNew) }
    }
  }
  e }
```

(foo + 42) / bar + arg



Not ambiguous, but gives wrong tree.

Ambiguous grammar: if some token sequence has multiple parse trees

(then it is has multiple abstract trees)

Two trees, each following the grammar, their leaves both give the same token sequence.

Exercise: Another Balanced Parenthesis Grammar

Show that the following balanced parentheses grammar is ambiguous (by finding two parse trees for some input sequence) and find unambiguous grammar for the same language.

$$B ::= \varepsilon \mid \textbf{(}\ B\ \textbf{)} \mid B\ B$$

Is this grammar ambiguous?

$$B ::= \varepsilon \mid \textbf{(}\ B\ \textbf{)}\ B$$

Exercise: Balanced Parentheses

Show that the following balanced parentheses grammar is ambiguous (by finding two parse trees for some input sequence) and find unambiguous grammar for the same language.

$B ::= \varepsilon \mid \mathbf{(}\ B\ \mathbf{)} \mid B\ B$

Remark

- The same parse tree can be derived using two different derivations, e.g.

  B -> (B) -> (BB) -> ((B)B) -> ((B)) -> (())

  B -> (B) -> (BB) -> ((B)B) -> (()B) -> (())

this correspond to different orders in which nodes in the tree are expanded

- Ambiguity refers to the fact that there are actually multiple *parse trees*, not just multiple derivations.

Towards Solution

- (Note that we must preserve precisely the set of strings that can be derived)
- This grammar:

B ::= ε | A

A ::= **( )** | A A | (A)

solves the problem with multiple ε symbols generating different trees, but it is still ambiguous: string **( ) ( ) ( )** has two different parse trees

Solution

- Proposed solution:

  B ::= ε | B **(**B**)**

- this is very smart! How to come up with it?
- Clearly, rule B::= B B generates any sequence of B's. We can also encode it like this:

  B ::= C*

  C ::= (B)

- Now we express sequence using recursive rule that does not create ambiguity:

  B ::= ε | C B

  C ::= (B)

- but now, look, we "inline" C back into the rules for so we get exactly the rule

  B ::= ε | B **(**B**)**

This grammar is not ambiguous and is the solution. We did not prove this fact (we only tried to find ambiguous trees but did not find any).

Exercise 2: Dangling Else

The dangling-else problem happens when the conditional statements are parsed using the following grammar.

S ::= S **;** S

S ::= id **:=** E

S ::= **if** E **then** S

S ::= **if** E **then** S else S

Find an unambiguous grammar that accepts the same conditional statements and matches the else statement with the nearest unmatched if.

Discussion of Dangling Else

```
if (x > 0) then
  if (y > 0) then
    z  = x + y
else x = - x
```

- This is a real problem languages like C, Java
  - resolved by saying **else** binds to innermost **if**
- Can we design grammar that allows all programs as before, but only allows parse trees where else binds to innermost if?

Sources of Ambiguity in this Example

- Ambiguity arises in this grammar here due to:
  - dangling **else**
  - binary rule for sequence (**;**) as for parentheses
  - priority between if-then-else and semicolon (**;**)

```
if (x > 0)
   if (y > 0)
     z  = x + y;
     u = z + 1      // last assignment is not inside if
```

Wrong parse tree -> wrong generated code

How we Solved It

We identified a wrong tree and tried to refine the grammar to prevent it, by making a copy of the rules. Also, we changed some rules to disallow sequences inside if-then-else and make sequence rule non-ambiguous. The end result is something like this:

S::= ε |A S // a way to write S::=A*
A ::= id **:=** E
A ::= **if** E **then** A
A ::= **if** E **then** A' **else** A

A' ::= id **:=** E
A' ::= **if** E **then** A' **else** A'

At some point we had a useless rule, so we deleted it.

We also looked at what a practical grammar would have to allow sequences inside if-then-else. It would add a case for blocks, like this:

A ::= **{** S **}**
A' ::= **{** S **}**

We could factor out some common definitions (e.g. define A in terms of A'), but that is not important for this problem.

Exercise: Unary Minus

1) Show that the grammar

A ::= − A
A ::= A − id
A ::= id

is ambiguous by finding a string that has two different syntax trees.

2) Make two different unambiguous grammars for the same language:

a) One where prefix minus binds stronger than infix minus.

b) One where infix minus binds stronger than prefix minus.

3) Show the syntax trees using the new grammars for the string you used to prove the original grammar ambiguous.

Exercise:
Left Recursive and Right Recursive

We call a production rule "left recursive" if it is of the form

A ::= A p

for some sequence of symbols p. Similarly, a "right-recursive" rule is of a form

A ::= q A

Is every context free grammar that contains both left and right recursive rule for a some nonterminal A ambiguous?

Answer: yes, if A is reachable from the top symbol and productive can produce a sequence of tokens

Making Grammars Unambiguous
- some recipes -

Ensure that there is always only one parse tree

Construct the correct abstract syntax tree

Goal: Build Expression Trees

```
abstract class Expr
case class Variable(id : Identifier) extends Expr
case class Minus(e1 : Expr, e2 : Expr) extends Expr
case class Exp(e1 : Expr, e2 : Expr) extends Expr
```

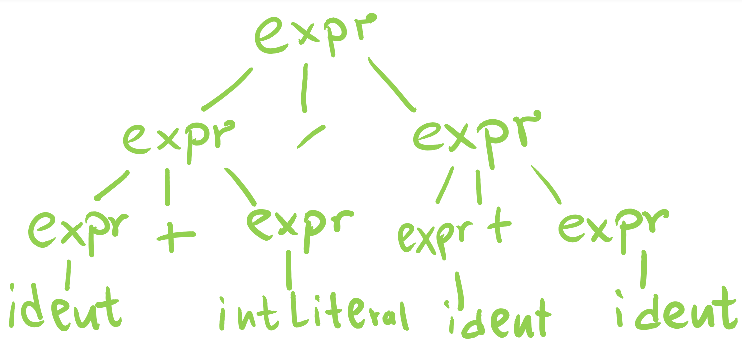
e1 - e2 - e3

different order gives different results:

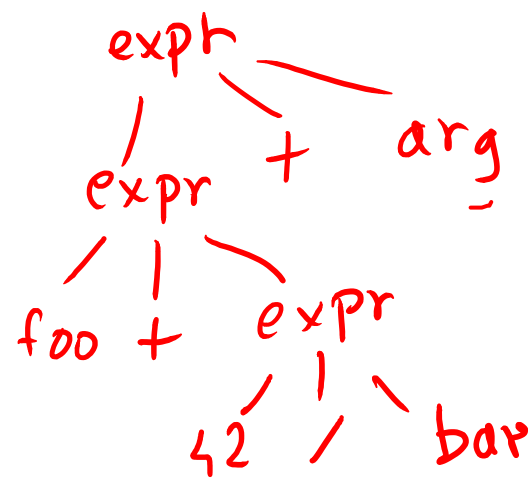
Minus(e1, Minus(e2,e3))	e1 - (e2 - e3)
Minus(Minus(e1,e2),e3)	(e1 - e2) - e3

Ambiguous Expression Grammar

expr ::= intLiteral | ident
 | expr + expr | expr / expr



foo + 42 / bar + arg



Each node in parse tree is given by one grammar alternative.

Show that the input above has two parse trees!

1) Layer the grammar by priorities

expr ::= ident | expr - expr | expr ^ expr | (expr)

expr ::= term (- term)*

term ::= factor (^ factor)*

factor ::= id | (expr)

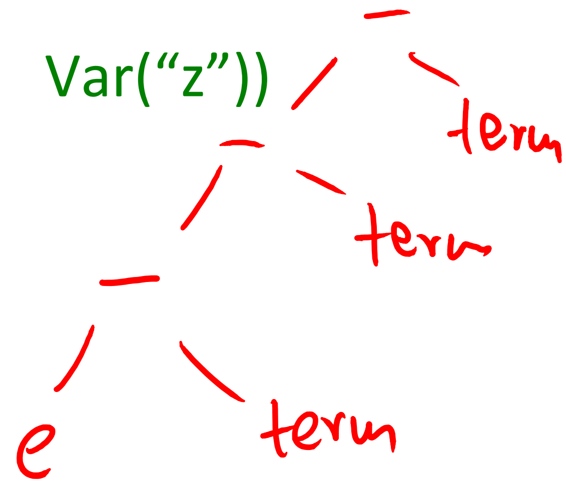
lower priority binds weaker, so it goes outside

2) Building trees: left-associative "-"

LEFT-associative operator

x – y – z ➔ (x – y) – z

Minus(Minus(Var(“x”),Var(“y”)), Var(“z”))




```
def expr : Expr = {
  var e = term
  while (lexer.token == MinusToken) {
    lexer.next
    e = Minus(e, term)
  }
  e
}
```

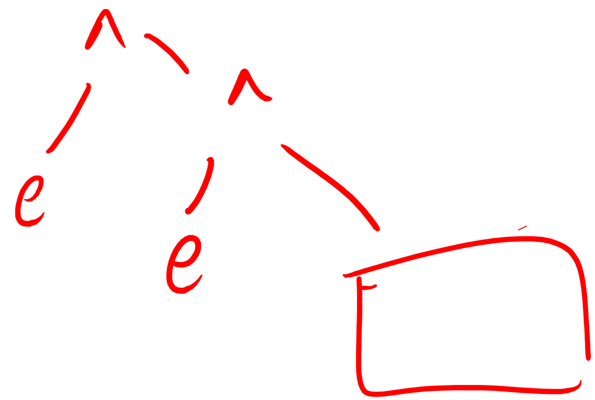
3) Building trees: right-associative "^"

RIGHT-associative operator – using recursion
(or also loop and then reverse a list)

x ^ y ^ z ➔ x ^ (y ^ z)

Exp(Var(“x”), Exp(Var(“y”), Var(“z”)))

```
def expr : Expr = {
  val e = factor
  if (lexer.token == ExpToken) {
    lexer.next
    Exp(e, expr)
  } else e
}
```



Manual Construction of Parsers

- Typically one applies previous transformations to get a nice grammar
- Then we write recursive descent parser as set of mutually recursive procedures that check if input is well formed
- Then enhance such procedures to construct trees, paying attention to the associativity and priority of operators

Grammar Rules as Logic Programs

Consider grammar G: S ::= a | b S

L(_) - language of non-terminal

L(G) = L(S) where S is the start non-terminal

L(S) = L(G) = { $b^n a$ | n >= 0}

From meaning of grammars:

$$w \in L(S) \Leftrightarrow w{=}a \bigvee w \in L(b\ S)$$

To check left hand side, we need to check right hand side. Which of the two sides?

- restrict grammar, use current symbol to decide - LL(1)
- use dynamic programming (CYK) for any grammar

Recursive Descent - LL(1)

- See wiki for
  - computing first, nullable, follow for non-terminals of the grammar
  - construction of parse table using this information
  - LL(1) as an interpreter for the parse table

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 )")
```

Rough General Idea

$A ::= B_1 \ldots B_p$
$\quad | \; C_1 \ldots C_q$
$\quad | \; D_1 \ldots D_r$

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

where:

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

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

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**

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

$$A ::= B_1 \ldots B_p \mid C_1 \ldots C_q \mid D_1 \ldots D_r$$

$\Rightarrow$

$$\mathrm{first}(A) = \mathrm{first}(B_1 \ldots B_p) \cup \mathrm{first}(C_1 \ldots C_q) \cup \mathrm{first}(D_1 \ldots D_r)$$

Sequences

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

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

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)

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

expr' = term'
termList' = {**+**}
U {**-**}

term' = factor'
factorList' = {*****}
U { **/** }

factor' = name' U { **(** }
name' = { **ident** }

nullable: termList, factorList

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

Example to Generate Constraints

$S ::= X \mid Y$
$X ::= \mathbf{b} \mid S\ Y$
$Y ::= Z\ X\ \mathbf{b} \mid Y\ \mathbf{b}$
$Z ::= \varepsilon \mid \mathbf{a}$

$S' = X' \cup Y'$
$X' =$

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

reachable (from S):
productive:
nullable:

First sets of terminals:
$S', X', Y', Z' \subseteq \{a,b\}$

Example to Generate Constraints

$S ::= X \mid Y$
$X ::= \mathbf{b} \mid S\ Y$
$Y ::= Z\ X\ \mathbf{b} \mid Y\ \mathbf{b}$
$Z ::= \varepsilon \mid \mathbf{a}$

$S' = X' \cup Y'$
$X' = \{b\} \cup S'$
$Y' = Z' \cup \mathbf{X'} \quad \cup Y'$
$Z' = \{a\}$

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

$S ::= X \mid Y$
$X ::= \mathbf{b} \mid S\ Y$
$Y ::= Z\ X\ \mathbf{b} \mid Y\ \mathbf{b}$
$Z ::= \varepsilon \mid \mathbf{a}$

$\Rightarrow$

$S' = X' \mid Y'$
$X' = 0 \mid (S' \,\&\, Y')$
$Y' = (Z' \,\&\, X' \,\&\, 0) \mid (Y' \,\&\, 0)$
$Z' = 1 \mid 0$

$S', X', Y', Z' \in \{0,1\}$

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

Rough General Idea

$A ::= B_1 \ldots B_p$
$\quad | \; C_1 \ldots C_q$
$\quad | \; D_1 \ldots D_r$

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

where:

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

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

Exercise 1

A ::= B **EOF**

B ::= ε | B B | **(**B**)**

nullable

A	B
0	0
0	1

- Tokens: **EOF**, **(**, **)**
- Generate constraints and compute nullable and first for this grammar.
- Check whether first sets for different alternatives are disjoint.

Exercise 2

S ::= B **EOF**
B ::= ε | B **(**B**)**

- Tokens: **EOF**, **(**, **)**
- Generate constraints and compute nullable and first for this grammar.
- Check whether first sets for different alternatives are disjoint.

Exercise 3

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
```

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

General Idea for nullable(A)

$$A ::= B_1 \ldots B_p \mid C_1 \ldots C_q \mid D_1 \ldots D_r$$

$\Rightarrow$

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 (e.g. second)
T1, T2, T3, T_F should be pairwise **disjoint** sets of tokens.

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)
- 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

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

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)**

nullable: B

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

follow(S) = {}

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

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

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

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]

$$\begin{array}{lll} A ::= & \textbf{(1)} & B_1 \ldots B_p \\ \mid & \textbf{(2)} & C_1 \ldots C_q \\ \mid & \textbf{(3)} & D_1 \ldots D_r \end{array}$$

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

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

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) = {1} 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

Transform Grammar for LL(1)

S ::= B **EOF**

B ::= ε | B **(**B**)**

(1) **(2)**

Old parsing table:

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

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))

Transform the grammar so that parsing table has no conflicts.

S ::= B **EOF**

B ::= ε | **(**B**)** B

(1) **(2)**

Left recursion is bad for LL(1)

	EOF	(	)
S			
B			

choice(A,t)

Parse Table is Code for Generic Parser

```
var stack : Stack[GrammarSymbol] // terminal or non-terminal
stack.push(EOF);
stack.push(StartNonterminal);
var lex = new Lexer(inputFile)
while (true) {
 X = stack.pop
  t = lex.curent
  if (isTerminal(X))
    if (t==X)   if (X==EOF) return success
                else lex.next // eat token t
    else parseError("Expected " + X)
  else { // non-terminal
    cs = choice(X)(t) // look up parsing table
    cs match { // result is a set
    case {i} => { // exactly one choice
      rhs = p(X,i) // choose correct right-hand side
      stack.push(reverse(rhs)) }
    case {} => parseError("Parser expected an element of " + unionOfAll(choice(X)))
    case _ => crash("parse table with conflicts - grammar was not LL(1)")
  }
}
```

What if we cannot transform the grammar into LL(1)?

1) Redesign your language

2) Use a more powerful parsing technique