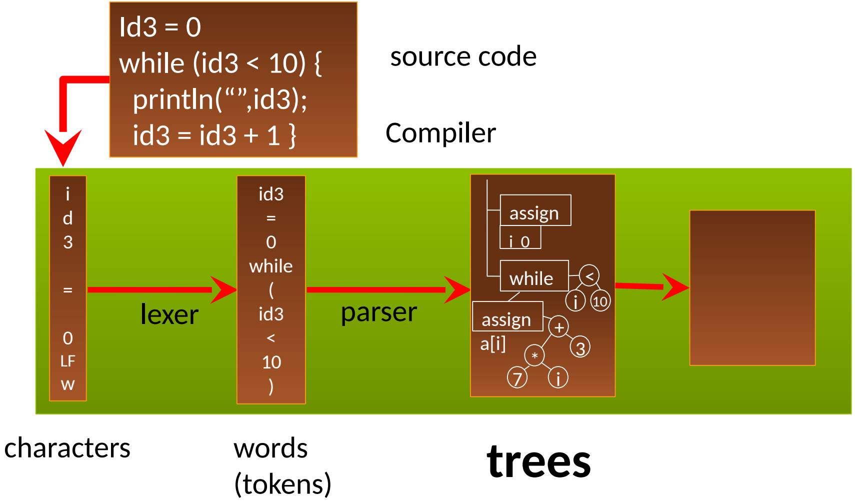
Parse Trees and Syntax Trees


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

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

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.

parse(prettyPrint(ast)) ≈ ast

Parse Tree vs Abstract Syntax Tree (AST)

- Each node in **parse tree** has children corresponding **precisely to right-hand side of grammar rules**. The definition of parse trees is fixed given the grammar
  - Often compiler never actually builds parse trees in memory
- Nodes in **abstract syntax tree (AST)** contain only useful information and usually omit the punctuation signs. We can choose our own syntax trees, to make it convenient for both construction in parsing and for later stages of our compiler or interpreter
  - **A compiler often directly builds AST**

Abstract Syntax Trees for Statements

grammar:

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

AST classes:

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

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

Ambiguous Grammars

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

ident + intLiteral / ident + ident

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).

Ambiguous Expression Grammar

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

Example input:

ident + intLiteral / ident

has two parse trees, one suggested by

ident   +   intLiteral / ident

and one by

ident + intLiteral   /   ident

Suppose Division Binds Stronger

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

Example input:

ident + intLiteral / ident

has two parse trees, one suggested by

ident + intLiteral / ident

and one by **a bad tree**

ident + intLiteral / ident

We do not want arguments of / expanding into expressions with **+** as the top level.

Layering the Grammar by Priorities

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

is transformed into a **new grammar**:

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

The bad tree

ident + intLiteral / ident

cannot be derived in the new grammar.

New grammar: same language, fewer parse trees!

Left Associativity of /

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

Example input:

ident / intLiteral / ident x/9/z

has two parse trees, one suggested by

ident / intLiteral / ident (x/9)/z

and one by **a bad tree**

ident / intLiteral / ident x/(9/z)

We do not want RIGHT argument of / expanding into expression with / as the top level.

Left Associativity - Left Recursion

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

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

No bad / trees
Still bad + trees

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

No bad trees.
Left recursive!

Left vs Right Associativity

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

Left associative
Left recursive,
so not LL(1).

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

Unique trees.
Associativity wrong.
No left recursion.

expr ::= divExpr exprSeq
exprSeq ::= + expr | ε
divExpr ::= factor divExprSeq
divExprSeq ::= / divExpr | ε
factor ::= intLiteral | ident

Unique trees.
Associativity wrong.
LL(1): easy to pick an alternative to use.