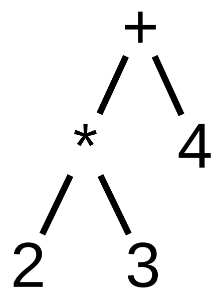
Parsing

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

2 * 3 + 4 $\Rightarrow$



Goal: discard irrelevant information to make it easier for the next stage.

Parentheses and most other forms of punctuation removed.

Grammars

Most programming languages described using a *context-free grammar*.

Compared to regular languages, context-free languages add one important thing: recursion.

Recursion allows you to count, e.g., to match pairs of nested parentheses.

Which languages do humans speak? I'd say it's regular: I do not not not not not not not not not not understand this sentence.

Languages

Regular languages (t is a terminal):

$$A \rightarrow t_1 \ldots t_n B$$

$$A \rightarrow t_1 \ldots t_n$$

Context-free languages (P is terminal or a variable):

$$A \rightarrow P_1 \ldots P_n$$

Context-sensitive languages:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2$$

"$B \rightarrow A$ only in the 'context' of $\alpha_1 \cdots \alpha_2$"

Issues

Ambiguous grammars

Precedence of operators

Left- versus right-recursive

Top-down vs. bottom-up parsers

Parse Tree vs. Abstract Syntax Tree

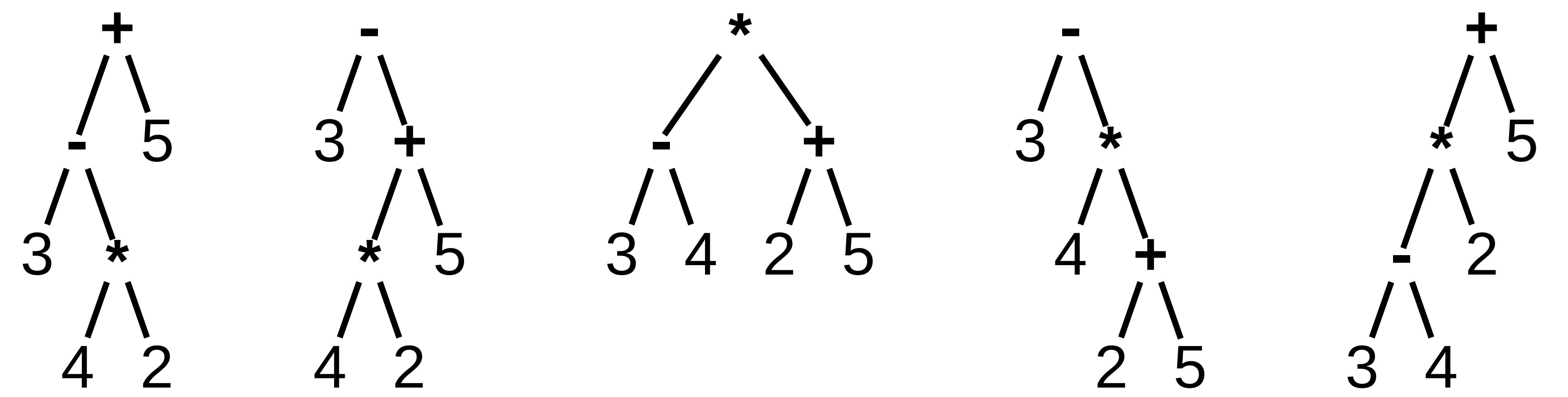
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

3 - 4 * 2 + 5

with the grammar

$$e \rightarrow e + e \mid e - e \mid e * e \mid e \,/\, e \mid N$$



Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

“My Dear Aunt Sally”

Mnemonic for multiplication and division before addition and subtraction.

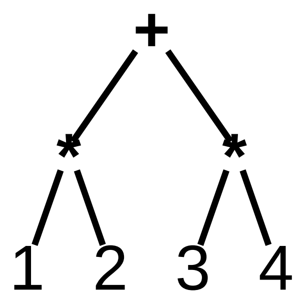
Operator Precedence

Defines how “sticky” an operator is.

`1 * 2 + 3 * 4`

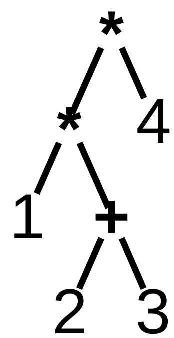
`*` at higher precedence than **`+`**:

`(1 * 2) + (3 * 4)`



`+` at higher precedence than **`*`**:

`1 * (2 + 3) * 4`



Associativity

Whether to evaluate left-to-right or right-to-left

Most operators are left-associative

`1 - 2 - 3 - 4`

```
        -
       / \
      -   4
     / \
    -   3
   / \
  1   2
```

((1 - 2) - 3) - 4

left associative

```
    -
   / \
  1   -
     / \
    2   -
       / \
      3   4
```

1 - (2 - (3 - 4))

right associative

Fixing Ambiguous Grammars

Original ANTLR grammar specification

```
expr
  : expr ’+’ expr
  | expr ’-’ expr
  | expr ’*’ expr
  | expr ’/’ expr
  | NUMBER
  ;
```

Ambiguous: no precedence or associativity.

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr ’+’ expr
     | expr ’-’ expr
     | term ;

term : term ’*’ term
     | term ’/’ term
     | atom ;

atom : NUMBER ;
```

Still ambiguous: associativity not defined

Assigning Associativity

Make one side or the other the next level of precedence

```
expr : expr '+' term
     | expr '-' term
     | term ;

term : term '*' atom
     | term '/' atom
     | atom ;

atom : NUMBER ;
```

Parsing LL(k) Grammars

LL: Left-to-right, Left-most derivation

k: number of tokens to look ahead

Parsed by top-down, predictive, recursive parsers

Basic idea: look at the next token to predict which production to use

ANTLR builds recursive LL(k) parsers

Almost a direct translation from the grammar.

Implementing a Top-Down Parser

```
stmt : 'if' expr 'then' expr
     | 'while' expr 'do' expr
     | expr ':=' expr ;
expr : NUMBER | '(' expr ')' ;
```

```
stmt() {
  switch (next-token) {
  case IF:
      match(IF); expr(); match(THEN); expr();          break;
  case WHILE:
      match(WHILE); expr(); match(DO); expr();         break;
  case NUMBER or LPAREN:
      expr(); match(COLEQ); expr();                    break;
}}
```

Writing LL(k) Grammars

Cannot have left-recursion

`expr : expr '+' term | term ;`

becomes

```
AST expr() {
  switch (next-token) {
  case NUMBER : expr(); /* Infinite Recursion */
```

Writing LL(1) Grammars

Cannot have common prefixes

```
expr : ID '(' expr ')'
     | ID '=' expr
```

becomes

```
expr() {
  switch (next-token) {
  case ID:
    match(ID); match(LPAR); expr(); match(RPAR);  break;
  case ID:
    match(ID); match(EQUALS); expr();                break;
```

Eliminating Common Prefixes

Consolidate common prefixes:

```
expr
  : expr '+' term
  | expr '-' term
  | term
  ;
```

becomes

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

Eliminating Left Recursion

Understand the recursion and add tail rules

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

becomes

```
expr : term exprt ;
exprt : '+' term exprt
      | '-' term exprt
      | /* nothing */
      ;
```

Using ANTLR’s EBNF

ANTLR makes this easier since it supports **`*`** and **`+`**:

```
expr : expr ’+’ term
     | expr ’-’ term
     | term ;
```

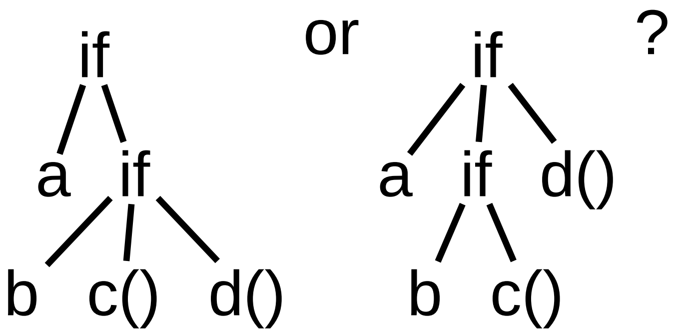
becomes

```
expr : term (’+’ term | ’-’ term)* ;
```

The Dangling Else Problem

Who owns the *else*?

```
if (a) if (b) c(); else d();
```



Grammars are usually ambiguous; manuals give disambiguating rules such as C’s:

> As usual the “else” is resolved by connecting an else with the last encountered elseless if.

The Dangling Else Problem

```
stmt : "if" expr "then" stmt iftail
     | other-statements ;

iftail
   : "else" stmt
   | /* nothing */
   ;
```

Problem comes when matching “iftail.”

Normally, an empty choice is taken if the next token is in the “follow set” of the rule. But since “else” can follow an iftail, the decision is ambiguous.

The Dangling Else Problem

ANTLR can resolve this problem by making certain rules "greedy." If a conditional is marked as greedy, it will take that option even if the "nothing" option would also match:

```
stmt
  : "if" expr "then" stmt
     ( options {greedy = true;}
     : "else" stmt
     )?
  | other-statements
  ;
```

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```
if a < b then a else b fi;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.

Statement separators/terminators

C uses `;` as a statement terminator.

```
if (a<b) printf("a less");
else {
  printf("b"); printf(" less");
}
```

Pascal uses `;` as a statement separator.

```
if a < b then writeln('a less')
else begin
  write('a'); writeln(' less')
end
```

Pascal later made a final `;` optional.