

Compiler Design

Lecture 7: Parse Tree and Abstract Syntax

Christophe Dubach

Winter 2025

Timestamp: 2025/01/31 14:53:00

A parser does more than simply recognising syntax. It performs certain actions such as:

- evaluate code (interpreter)
- emit code (simple compiler)
- build an internal representation of the program (multi-pass compiler)

In the case of

- a hand-written recursive descent parser:
integrate the actions with the parsing functions
- an automatically generated parser:
add actions to the grammar

Syntax Tree

In a multi-pass compiler, the parser builds a syntax tree which is used by the subsequent passes.

A syntax tree can be:

- a **concrete syntax tree**, or **parse tree**, if it directly corresponds to the context-free grammar
- an **abstract syntax tree** if it corresponds to a simplified — or abstract — grammar

The abstract syntax tree (AST) is usually used in compilers.

Table of contents

Parse Tree

- Ambiguous Grammars: nesting of structures

- Ambiguous Grammars: operator precedence

Abstract Syntax Tree

- Abstract Syntax

- Abstract Syntax Tree Representation

- AST Construction

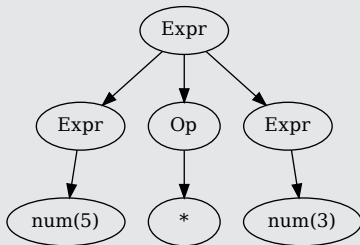
Parse Tree

Example: Concrete Syntax Tree (Parse Tree)

$\text{Expr} ::= \text{Expr Op Expr} \mid \text{num} \mid \text{id}$

$\text{Op} ::= + \mid *$

Concrete Syntax Tree for $5 * 3$



Ambiguity definition

- If a grammar has more than one leftmost (or rightmost) derivations for a single sentential form, the grammar is **ambiguous**
- This is a problem when interpreting an input program or when building an internal representation

Parse Tree

Ambiguous Grammars: nesting of
structures

Example: Dangling-Else

Ambiguous grammar

```
Stmt ::= if Expr then Stmt  
       | if Expr then Stmt else Stmt  
       | NonIfStmt
```

input

```
if E1 then if E2 then S1 else S2
```

One possible interpretation

```
if E1 then  
  if E2 then  
    S1  
else  
  S2
```

Another possible interpretation

```
if E1 then  
  if E2 then  
    S1  
else  
  S2
```

Removing Ambiguity

Must rewrite the grammar to avoid generating the problem:

- By adding a keyword to “end” the **if**:
 - e.g. Bash: `if ... fi`
 - e.g. Ada: `if ... end if;`
- By forcing the programmer to align the **else**:
 - e.g. Python

```
if cond1:
    if cond2:
        ...
else:
    ...
```

- Or by keeping the same language syntax and add extra rules in the grammar to match, for instance, each `else` to innermost unmatched `if` (common sense).

Let's try to keep the same language syntax and modify the grammar.

Unambiguous grammar

```
Stmt      ::= if Expr then Stmt  
           | if Expr then WithElse else Stmt  
           | NonIfStmt  
WithElse  ::= if Expr then WithElse else WithElse  
           | NonIfStmt
```

- Intuition: the `WithElse` restricts what can appear in the `then` part
- With this grammar, the example has only one derivation

Unambiguous grammar for dangling-else

```
Stmt      ::= if Expr then Stmt  
            | if Expr then WithElse else Stmt  
            | NonIfStmt  
WithElse  ::= if Expr then WithElse else WithElse  
            | NonIfStmt
```

Derivation for: if E1 then if E2 then S1 else S2

```
Stmt  
if Expr then Stmt  
if E1 then Stmt  
if E1 then if Expr then WithElse else Stmt  
if E1 then if E2 then WithElse else Stmt  
if E1 then if E2 then S1 else Stmt  
if E1 then if E2 then S1 else S2
```

This binds the else controlling S2 to the inner if.

Beware: Dangling-Else and Recursive Descent Parser

Unambiguous grammar for dangling-else

```
Stmt      ::= if Expr then Stmt  
            | if Expr then WithElse else Stmt  
            | NonIfStmt  
WithElse  ::= if Expr then WithElse else WithElse  
            | NonIfStmt
```

⚠ This grammar is not LL(k) ⚠

A recursive descent parser should use the original grammar:

Ambiguous grammar

```
Stmt ::= if Expr then Stmt [else Stmt]  
      | NonIfStmt
```

This works because when the recursive descent parser parses an **if** statement, it will always consume as much of the input as possible. Therefore, an **else** will always be associated with the nearest **then**.

Parse Tree

Ambiguous Grammars: operator
precedence

Example : Arithmetic Operators

Ambiguous Grammar

Expr ::= Expr Op Expr | num | id

Op ::= + | *

This grammar has multiple leftmost derivations for $x + 2 * y$

One possible derivation

Expr

Expr Op Expr

id(x) Op Expr

id(x) + Expr

id(x) + Expr Op Expr

id(x) + num(2) Op Expr

id(x) + num(2) * Expr

id(x) + num(2) * id(y)

$x + (2 * y)$

Another possible derivation

Expr

Expr Op Expr

Expr Op Expr Op Expr

id(x) Op Expr Op Expr

id(x) + Expr Op Expr

id(x) + num(2) Op Expr

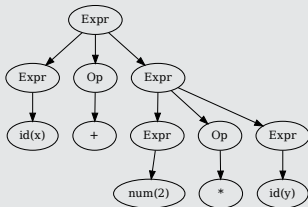
id(x) + num(2) * Expr

id(x) + num(2) * id(y)

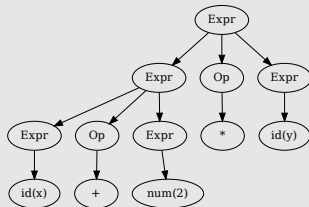
$(x + 2) * y$

Parse trees

$x + (2 * y)$



$(x + 2) * y$



Which one is correct?

As far as the grammar is concerned: both!

However, they represent different expressions!

Encoding Precedence

Some operators have higher precedence over others.

For instance, multiplication takes precedence over addition:

- $x + 2 * y = x + (2 * y)$
- $x * 2 + y = (x * 2) + y$

Grammar for basic arithmetic expressions

```
Expr ::= Expr Op Expr | num | id | '(' Expr ')'  
Op    ::= '+' | '-' | '*' | '/'
```

Perhaps we can massage this grammar so that $+$ and $-$ have lower precedence than $*$ and $/$?

Grammar for basic arithmetic expressions

```
Expr ::= Expr Op Expr | num | id | '(' Expr ')'  
Op    ::= '+' | '-' | '*' | '/'
```

Main idea: introduce one non-terminal symbol for every precedence level. One for +, −, and one for * and / .

Grammar with precedence encoded

```
Expr    ::= Term ( ('+' | '-') Term )  
Term    ::= Factor ( ('*' | '/') Factor )  
Factor  ::= number | id | '(' Expr ')'
```

Let's remove the EBNF syntax

Example: CFG for arithmetic expressions (EBNF form)

```
Expr    ::= Term ( ( '+' | '-' ) Term )*  
Term    ::= Factor ( ( '*' | '/' ) Factor )*  
Factor  ::= number | id | '(' Expr ')'
```

After removal of EBNF syntax

```
Expr     ::= Term Terms  
Terms    ::= ( '+' | '-' ) Term Terms |  $\epsilon$   
Term     ::= Factor Factors  
Factors  ::= ( '*' | '/' ) Factor Factors |  $\epsilon$   
Factor   ::= number | id | '(' Expr ')'
```

After further simplification

```
Expr     ::= Term ( ( '+' | '-' ) Expr |  $\epsilon$  )  
Term     ::= Factor ( ( '*' | '/' ) Term |  $\epsilon$  )  
Factor   ::= number | id | '(' Expr ')'
```

Concrete Syntax Tree (Parse Tree)

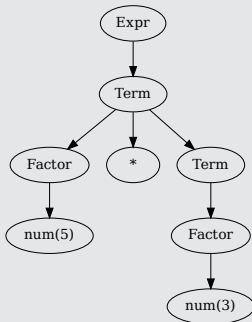
CFG for arithmetic expressions

Expr ::= Term (('+' | '-') Expr | ϵ)

Term ::= Factor (('*' | '/') Term | ϵ)

Factor ::= number | id | '(' Expr ')'

Concrete Syntax Tree for 5 * 3



Deeper ambiguity

- Ambiguity usually refers to confusion in the CFG (Context Free Grammar)
- Consider the following case: $a = f(17)$
In some languages (e.g. Algol, Scala), f could be either a **function** of an **array**
- In such case, context is required
 - Need to track declarations
 - Really a type issue, not context-free syntax
 - Requires an extra-grammatical solution
 - Must handle these with a different mechanism

Solution:

Step outside the grammar rather than making it more complex.
This will be treated during semantic analysis.

Ambiguity Final Words

Ambiguity arises from two distinct sources:

- Confusion in the context-free syntax
e.g. `if then else`
- Confusion that requires context to be resolved
e.g. `array vs function`

Resolving ambiguity:

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity delay the detection of such problem (semantic analysis phase)
 - For instance, it is legal during syntactic analysis to have:
`void i; i=4;`

Abstract Syntax Tree

Abstract Syntax Tree

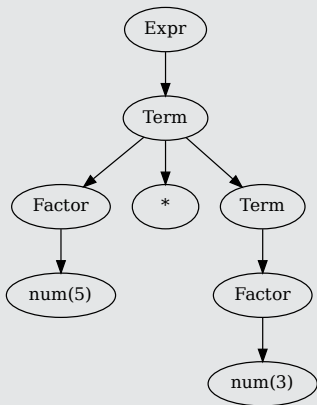
Abstract Syntax

Concrete Syntax Tree (Parse Tree)

CFG for arithmetic expressions

Expr ::= Term (('+' | '-') Expr | ϵ)
Term ::= Factor (('*' | '/') Term | ϵ)
Factor ::= number | id | '(' Expr ')'

Concrete Syntax Tree for 5 * 3



The concrete syntax tree contains a lot of unnecessary information.

It is possible to simplify the concrete syntax tree to remove the redundant information.

```
Expr    ::= Term (('+' | '-' ) Expr |  $\epsilon$ )  
Term    ::= Factor (('*' | '/' ) Term |  $\epsilon$ )  
Factor  ::= number | id | '(' Expr ')'
```

Exercise

1. Write the concrete syntax tree for $3 * (4 + 5)$
2. Simplify the tree.

Abstract Grammar

These simplifications leads to a new simpler context-free grammar called **Abstract Grammar**.

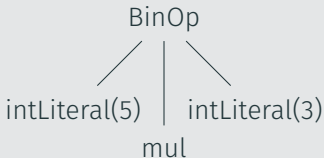
Example: abstract grammar for arithmetic expressions

Expr ::= BinOp | intLiteral | id

BinOp ::= Expr Op Expr

Op ::= add | sub | mul | div

5 * 3



This is called an **Abstract Syntax Tree**

Example: abstract grammar for arithmetic expressions

```
Expr ::= BinOp | intLiteral | id  
BinOp ::= Expr Op Expr  
Op      ::= add | sub | mul | div
```

Note that for given concrete grammar, there exist more than one abstract grammar. For instance:

```
Expr ::= AddOp | SubOp | MulOp | DivOp | intLiteral | id  
AddOp ::= Expr add Expr  
SubOp ::= Expr sub Expr  
MulOp ::= Expr mul Expr  
DivOp ::= Expr div Expr
```

We pick the most suitable grammar for the compiler.

Abstract Syntax Tree

Abstract Syntax Tree Representation

Abstract Syntax Tree using Object Oriented Design

The Abstract Syntax Tree (AST) forms the main intermediate representation of the compiler's front-end.

$\forall$ non-terminal/terminal in the abstract grammar, define a class.

- If a non-terminal has any alternative on the rhs (right hand side), then the class is abstract (cannot instantiate it).
The terminal or non-terminal appearing on the rhs are subclasses of the non-terminal on the lhs.
- The sub-trees are represented as fields in the class.
- Each non-abstract class has a unique constructor.
- If a terminal does not store any information, then we can use an Enum type in Java instead of a class.

Example: abstract grammar for arithmetic expressions

```
Expr ::= BinOp | intLiteral | id
BinOp ::= Expr Op Expr
Op      ::= add | sub | mul | div
```

Corresponding Java Classes

```
abstract class Expr { }

class IntLiteral extends Expr {
    int i;
    IntLiteral(int i){...}
}

class Id extends Expr {
    String name;
    Id(String name){...}
}

class BinOp extends Expr {
    Op op;
    Expr lhs;
    Expr rhs;
    BinOp(Op op, Expr lhs, Expr rhs) {...}
}

enum Op {ADD, SUB, MUL, DIV}
```

Abstract Syntax Tree

AST Construction

How to build the AST?

Two main approaches:

- Build the concrete syntax tree during parsing, and then use a pass to *transform* the concrete syntax tree into an abstract syntax tree using local transformation rules recursively.
- Directly build the AST during parsing without going through the concrete syntax tree.

Approach 1: Concrete Syntax Tree to AST

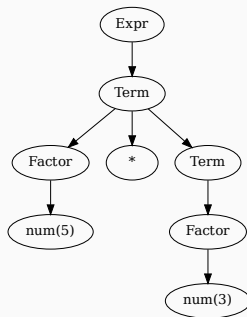
CFG for arithmetic expressions

Expr ::= Term (('+' | '-') Expr | ϵ)
Term ::= Factor (('*' | '/') Term | ϵ)
Factor ::= number | id | '(' Expr ')'

Concrete syntax tree for 5×3

Transformation rules

Expr(t, op, e) $\rightarrow$ BinOp(t, op, e)
Expr(t) $\rightarrow$ t
Term(f, op, t) $\rightarrow$ BinOp(f, op, t)
Term(x) $\rightarrow$ x
Factor(Num(x)) $\rightarrow$ intLiteral(x)
Factor(id(x)) $\rightarrow$ id(x)
Factor('(' , e, ')') $\rightarrow$ e



Apply the transformations depth-first on the concrete syntax tree.

Approach 2: Building the AST during parsing

CFG for arithmetic expressions

```
Expr    ::= Term (('+' | '-') Expr |  $\epsilon$ )
Term    ::= Factor (('*' | '/' ) Term |  $\epsilon$ )
Factor  ::= number | id | '(' Expr ')'
```

Current Parser (class)

```
void parseExpr() {
    parseTerm();
    if (accept(PLUS|MINUS))
        nextToken();
        parseExpr();
}

void parseTerm() {
    parseFactor();
    if (accept(TIMES|DIV))
        nextToken();
        parseTerm();
}

void parseFactor() {
    if (accept(LPAR))
        nextToken();
        parseExpr();
        expect(RPAR);
    else
        expect(NUMBER|ID);
}
```

Let's modify our existing parser to produce the AST:

- Each `parseXXX()` function will now return an AST node.
- We assume that **expect** now returns the token.

Current Parser

```
void parseExpr() {  
    parseTerm();  
    if (accept(PLUS|MINUS))  
        nextToken();  
    parseExpr();  
}
```

AST building (modified Parser)

```
Expr parseExpr() {  
    Expr lhs = parseTerm();  
    if (accept(PLUS|MINUS))  
        Op op;  
        if (token == PLUS)  
            op = ADD;  
        else // token == MINUS  
            op = SUB;  
        nextToken();  
    Expr rhs = parseExpr();  
    return new BinOp(op, lhs, rhs);  
    return lhs;  
}
```

Current Parser

```
void parseTerm() {  
    parseFactor();  
    if (accept(TIMES|DIV))  
        nextToken();  
    parseTerm();  
}
```

AST building (modified Parser)

```
Expr parseTerm() {  
    Expr lhs = parseFactor();  
    if (accept(TIMES|DIV))  
        Op op;  
        if (token == TIMES)  
            op = MUL;  
        else // token == DIV  
            op = DIV;  
        nextToken();  
    Expr rhs = parseTerm();  
    return new BinOp(op, lhs, rhs);  
    return lhs;  
}
```

Current Parser

```
void parseFactor() {  
    if (accept(LPAR))  
        nextToken();  
        parseExpr();  
        expect(RPAR);  
    else  
        expect(NUMBER|ID);  
}
```

AST building (modified Parser)

```
Expr parseFactor() {  
    if (accept(LPAR))  
        nextToken();  
        Expr e = parseExpr();  
        expect(RPAR);  
        return e;  
    else if (accept(NUMBER))  
        IntLiteral il = parseNumber();  
        return il;  
    else if (accept(ID))  
        Id id = parseId();  
        return id;  
    else  
        expect(LPAR|NUMBER|ID)  
}  
  
IntLiteral parseNumber() {  
    Token n = expect(NUMBER);  
    int i = Integer.parseInt(n.data);  
    return new IntLiteral(i);  
}  
  
Id parseId() {  
    Token t = expect(IDENT);  
    return new Id(t.data);  
}
```

Beware: Associativity

What is the result of $2 - 3 - 4$?

- -5

Why? Mathematical operators are usually left-associative:

- $2 - 3 - 4 = (2 - 3) - 4 = -1 - 4 = -5$

How does our parser handle $2 - 3 - 4$?

Parser code (partial)

```
Expr parseExpr() {
    Expr lhs = parseTerm();
    if (accept(PLUS|MINUS))
        Op op;
        if (token == PLUS) op = ADD;
        else op = SUB;
        nextToken();
        Expr rhs = parseExpr();
        return new BinOp(op, lhs, rhs);
    return lhs;
}

Expr parseTerm() {
    Expr lhs = parseFactor();
    if (accept(TIMES|DIV))
        ...
    return lhs;
}

Expr parseFactor() {
    if (accept(LPAR)) ...
    else if (accept(NUMBER))
        IntLiteral il = parseNumber();
        return il;
    else ...
}
```

Exercise: draw the AST

 The resulting tree is right-associative!

How to solve this associativity problem?

Three solutions:

- Build the concrete syntax tree, and then use a pass to change nodes order.
- Modify the grammar.
- Modify the parser to directly produce the correct AST.

Let's try to modify our grammar

Grammar with right-associative operators

```
Expr    ::= Term (('+' | '-' ) Expr |  $\epsilon$ )  
Term    ::= Factor (('*' | '/' ) Term |  $\epsilon$ )  
Factor  ::= number | id | '(' Expr ')'
```

Equivalent grammar with right-associative operators

```
Expr    ::= Term ('+' | '-' ) Expr | Term  
Term    ::= Factor ('*' | '/' ) Term | Factor)  
Factor  ::= number | id | '(' Expr ')'
```

Grammar with left-associative operators

```
Expr    ::= Expr ('+' | '-' ) Term | Term  
Term    ::= Term ('*' | '/' ) Factor | Factor)  
Factor  ::= number | id | '(' Expr ')'
```

Can you see the problem?

Making our parser left-associative

Right-associative

```
Expr parseExpr() {  
    Expr lhs = parseTerm();  
    if (accept(PLUS|MINUS))  
        Op op;  
        if (token == PLUS)  
            op = ADD;  
        else  
            op = SUB;  
        nextToken();  
        Expr rhs = parseExpr();  
        return new BinOp(op, lhs, rhs);  
    return lhs;  
}  
...
```

Left-associative

```
Expr parseExpr() {  
    Expr lhs = parseTerm();  
    while (accept(PLUS|MINUS))  
        Op op;  
        if (token == PLUS)  
            op = ADD;  
        else  
            op = SUB;  
        nextToken();  
        Expr rhs = parseTerm();  
        lhs = new BinOp(op, lhs, rhs);  
    return lhs;  
}  
...
```

Recursion replaced by iteration.

Why does it work?

Grammar implemented by the “right-associative” parser

```
Expr    ::= Term (( '+' | '-' ) Expr |  $\epsilon$ )  
Term    ::= Factor (( '*' | '/' ) Term |  $\epsilon$ )  
Factor  ::= number | id | '(' Expr ')'
```

Remember where we came from:

EBNF form

```
Expr    ::= Term ( ( '+' | '-' ) Term )*  
Term    ::= Factor ( ( '*' | '/' ) Factor )*  
Factor  ::= number | id | '(' Expr ')'
```

Putting it all together

Steps to produce a parser that builds a correct AST:

1. Start with the grammar and encode precedence;
2. Remove any left-recursion if present;
3. Pay attention to associativity of operators
 - For left-associative operators, use Kleen-closure (EBNF), this will lead to a loop in the parser
 - For right-associative operators, remove any Kleen-closure
4. Write recursive descent parser following the grammar and ensure left-associative operators are handled properly.

Next lecture

- AST processing