Ambiguous and unambiguous grammars

Ambiguous grammars for defining the syntax of languages

- ambiguous grammars are simpler and more readable
- the syntax of a language is usually defined by
  - an ambiguous grammar
  - rules for syntactic associativity and precedence

Unambiguous grammars for implementing parsers

- a parser driven by an unambiguous grammar “knows” that for each token there is at most one applicable production

Standard techniques for grammar disambiguation

Problem

- a grammar G is ambiguous but we do not want to change the language defined by G (example: expressions with infix operators)
- instead, we define syntactic associativity and precedence rules to get unique derivation trees
- can we define a non-ambiguous grammar for the same language to include syntactic associativity and precedence rules?

Possible solution

- transform G into an equivalent non-ambiguous grammar G'
- equivalent means that for all non-terminal symbols B of G, the language generated by G and G' from B is the same
- the transformation is driven by the syntactic associativity and precedence rules

Example 1: + and $\star$ with the same precedence

Ambiguous grammar

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Non-ambiguous grammar, left associative operations

```
Exp ::= Atom | Exp '+' Atom | Exp '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Non-ambiguous grammar, right associative operations

```
Exp ::= Atom | Atom '+' Exp | Atom '*' Exp
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

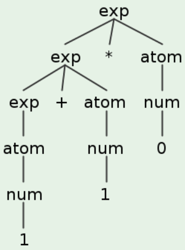
Example 1: $+$ and $\star$ with the same precedence

Non-ambiguous grammar, left associative operations

```
Exp ::= Atom | Exp '+' Atom | Exp '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Solution: In `Atom` expressions can contain $+$ or $\star$ only between parentheses

Unique derivation tree for `1+1*0`



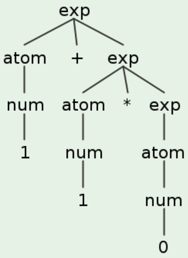
Example 1: $+$ and $\star$ with the same precedence

Non-ambiguous grammar, right associative operations

```
Exp ::= Atom | Atom '+' Exp | Atom '*' Exp
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Solution: In `Atom` expressions can contain $+$ or $\star$ only between parentheses

Unique derivation tree for `1+1*0`



Example 2: $\star$ with higher precedence

Ambiguous grammar

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Non-ambiguous grammar, left associative operations

```
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Non-ambiguous grammar, right associative operations

```
Exp ::= Mul | Mul '+' Exp
Mul ::= Atom | Atom '*' Mul
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

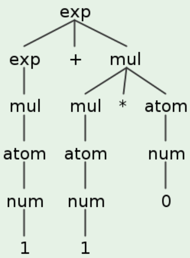
Example 2: $\star$ with higher precedence

Non-ambiguous grammar, left associative operations

```
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Solution: In `Mul` expressions can contain + only between parentheses
In `Atom` expressions can contain + or $\star$ only between parentheses

Unique derivation tree for `1+1*0`



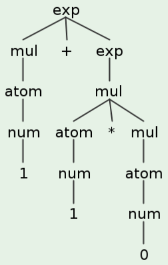
Example 2: $\star$ with higher precedence

Non-ambiguous grammar, right associative operations

```
Exp ::= Mul | Mul '+' Exp
Mul ::= Atom | Atom '*' Mul
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Solution: In `Mul` expressions can contain + only between parentheses
In `Atom` expressions can contain + or $\star$ only between parentheses

Unique derivation tree for `1+1*0`



Remaining examples

Non-ambiguous grammars can be easily defined by symmetry

- $\star$ higher precedence and left associative, $+$ right associative
- $\star$ higher precedence and right associative, $+$ left associative
- $+$ higher precedence and left associative, $\star$ left associative
- $+$ higher precedence and left associative, $\star$ right associative
- $+$ higher precedence and right associative, $\star$ left associative
- $+$ higher precedence and right associative, $\star$ right associative

Programming paradigms

Definition of programming paradigm

The programming style based on an emerging computational model

Main examples of paradigms

- imperative (the von Neumann style)
  based on the notions of instruction and state
  - procedural (example: C)
  - object-oriented (examples: C#,C++, Java, JavaScript, Python,. . .)
- declarative (based on a more abstract model)
  - functional (examples: Haskell,ML,...)
    based on the notions of mathematical function and function application
  - logic (example: Prolog)
    based on the notions of logic rule and query

Programming paradigms

Multi-paridigm programming languages

Modern programming languages embrace several paradigms to favor flexibility

Examples

C#,C++, Java, JavaScript, Python and others support both

- the imperative paradigm (mainly object-oriented, but also procedural)
- the declarative paradigm (mainly functional)

Purely functional paradigm

In a nutshell

- program=definitions of mathematical functions + a main expression
- computation=function application (what is called *function call* in an imperative context)
- no state: no variable assignment, more in general, no statements, just expressions
- variables=function parameters or “variables” storing constant values

Functions are first class values

Functions are obtained as the result of some types of expressions

Terminology

- higher order functions: functions that can accept functions as arguments or/and can return functions as result
- lambda expressions/functions or anonymous functions: functions defined by an expression

Languages and functional programming

Examples of languages considered primarily functional

- LISP (first functional languages, late 50s)
- ML (early 70s) and its family (OCaml, F#)
- Scheme (mid 70s, derived from LISP)
- Haskell (early 90s, purely functional)
- Clojure (2007, derived from LISP)

Most languages support functional programming

- C++
- C#
- Java
- JavaScript
- Kotlin
- Scala
- Python …

FP for beginners

Are there functional languages suitable for beginners?

Hard to tell ...

- Most mainstream languages support functional programming. However
  - not all typical features are supported. Example: pattern matching
  - the functional features cannot be easily isolated
- There are languages with better learning curve, although not mainstream

Why learning functional programming

- All mainstream languages and libraries based on functional features
- Functional features well-suited for several programming styles:
  - generic programming for code reuse and maintenance
  - event-driven programming (example: JavaScript/Node.js)
  - concurrent programming (example: Erlang)

OCaml

What is OCaml?

- French dialect of ML (1996)
- Multi-paradigm language with a purely functional core
- Statically typed with type inference
  - typing rules checked statically
  - types are automatically inferred (=deduced) and can be omitted in programs

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*

DecOrExp ::= Dec | Exp

Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp

Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'

Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Extended BNF (EBNF) grammars

- BNF is extended with the regular expressions operators `*`, `+`, `?`
- Example: `Pat+` means `Pat` concatenated one or more times
- Remark: `+` (reg-exp operator) is different from `'+'` (terminal symbol)

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Quick comments

- `ID` variable identifiers `(_[\w']|[a-zA-Z])[\w']*`
- `NUM` natural numbers
  `0[bB][01][01_]*|0[oO][0-7][0-7_]*|0[xX][\da-fA-F][\da-fA-F_]*|\d[\d_]*`
- `UOP` unary arithmetic operators `[+-]`
- `BOP` binary arithmetic operators `[+-*/]|mod`
- `Pat` patterns: very simple for now, a more complete definition will be considered later on

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Functions and application

- examples of functions

```
let inc = fun x -> x+1 (* the increment function *)
let inc2 x = x+1 (* a more compact syntax *)
```

- function application (= function call)

```
inc 3 (* syntax inc(3) optional, evaluation returns 4 *)
inc2 3 (* syntax inc2(3) optional, evaluation returns 4 *)
```

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Functions and application

- examples of anonymous function

```
fun x -> x+1 (* the increment function *)
```

- function application (= function call)

```
(fun x -> x+1) 3 (* evaluation returns 4 *)
```

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Semantics of function application

`exp1 exp2`

- `exp1` is evaluated in a function *f*
- `exp2` is evaluated in the argument *a* of *f*
- `exp1 exp2` is evaluated in $f(a)$ (*f* applied to *a*)

OCaml core

Precedence and associativity rules

- standard rules for arithmetic expressions
- application has higher precedence then binary operators

```
inc 1+2 (* equivalent to (inc 1)+2 *)
1+inc 2 (* equivalent to 1+(inc 2) *)
```

- anonymous functions have lower precedence

```
fun x->x+1 (* equivalent to fun x->(x+1) *)
fun f->f 2 (* equivalent to fun f->(f 2), not (fun f->f) 2 *)
```

- more critical cases: application and unary operators

```
inc + 3 (* addition *)          inc (+3) (* application *)
inc - 3 (* subtraction *)       inc (-3) (* application *)
+ inc 3 (* is +(inc 3) *)       - inc 3    (* is -(inc 3) *)
```

OCaml type inference

A simple interpreter session (Read Eval Print Loop)

Types can be automatically deduced (=inferred) by the interpreter!

```
# 42
- : int = 42
# fun x->x+1
- : int -> int = <fun>
# (fun x->x+1) 2
- : int = 3
```

Simplified syntax of OCaml core type expressions

BNF Grammar

```
Type ::= 'int' | Type '->' Type | '(' Type ')'
```

OCaml core types

Terminology

- `int` is a built-in simple type: the type of integers
- `int -> int` is a built-in composite type
- `->` is a type constructor: it is used for building composite types from simpler types
- types built with the `->` (arrow) constructor are called *arrow types* or *function types*

Meaning of arrow types

t_1 `->` t_2 is the type of functions from t_1 to t_2 that

- can only be applied to a single argument of type t_1
- always returns values of type t_2

OCaml core types

Remarks

- the arrow type constructor is right-associative

  `int->int->int` $=$ `int->(int->int)`

- a type constructor always builds a type different from its type components

  $t_1 \neq t_1\texttt{->}t_2$ and $t_2 \neq t_1\texttt{->}t_2$

- two arrow types are equal if they are built with the same type components

  $t_1\texttt{->}t_2 = t$ if and only if $t = t_3\texttt{->}t_4$, $t_3 = t_1$, $t_4 = t_2$

- Remark: from the items above

  `int->(int->int)` $\neq$ `(int->int)->int`

Types and type expressions

Remarks

- `int->int->int` is a type expressions, but is also called a type, because it represents a specific type
- `int->int->int` and `int->(int->int)` are different type expressions which represent the same type