Context free (CF) grammars

In a nutshell

- the most widespread formalism for defining the syntax of a PL
- more expressive than regular expressions
  - basic operators: concatenation and union
  - difference w.r.t. regular expressions: it is possible to use *names* and *recursive (=inductive) definitions*

Example of BNF grammar (Backus-Naur Form or Backus Normal Form)

A CF grammar for simple expressions

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

Remark

- `Num` is defined in the grammar only for completeness
- In practice, tokens as `Num` are defined separately by a regular expression

Context free (CF) grammars

Revisited example

```
Exp ::= NUM | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
```

`NUM` is defined by `0|1`

Notation

- in `Exp` only the first letter is capitalized: it is defined in the grammar
- in `NUM` all letters are capitalized: it is defined separately by a regular expression

Terminology of (CF) grammars

Example

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

Terminology: grammar $G = (T, N, P)$

- $\{$`'+'`, `'*'`, `'('`, `')'`, `'0'`, `'1'`$\}$ is the set T of terminal symbols
- $\{$`Exp`, `Num`$\}$ is the set N of non-terminal symbols
- $\{$(`Exp`,`Num`), (`Exp`,`Exp '+' Exp`), (`Exp`,`Exp '*' Exp`), (`Exp`,`'(' Exp ')'`), (`Num`,`'0'`), (`Num`,`'1'`)$\}$ is the set P of productions

Remarks

- each non terminal corresponds to a language; languages are defined as unions of concatenations
- terminal symbols are lexemes of the languages defined by the grammar
- productions have shape (B, α) where $B \in N$ and $\alpha \in (T \cup N)^*$

Grammars as inductive definitions of languages

Example

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

Inductive definition of languages

$Exp = Num \cup (Exp \cdot \{\texttt{"+"}\} \cdot Exp) \cup (Exp \cdot \{\texttt{"*"}\} \cdot Exp) \cup (\{\texttt{"("}\} \cdot Exp \cdot \{\texttt{")"}\})$
$Num = \{\texttt{"0"}\} \cup \{\texttt{"1"}\}$

Remarks

- $Exp = Num \cup \ldots$ is the base case for *Exp*: a number is an expression
- *Exp* is defined on top of *Num*, *Num* is defined only by base cases

Grammars as inductive definitions of languages

Another example

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

Remarks

The definitions of *Exp* and *Term* are *mutually recursive*

Derivations

Grammar

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

Languages generated by a grammar

- A grammar *generates* a language for each non-terminal symbol
- The grammar above generates the two languages L_{Exp} and L_{Num}
- The language for `Num` is pretty simple: $L_{Num} = \{\texttt{"0"},\texttt{"1"}\}$

Questions

- How is L_{Exp} defined?
- How can we show that $\texttt{"1+0"} \in L_{Exp}$ and $\texttt{"1+*("} \notin L_{Exp}$

Answer: one-step and multi-step *derivations* are used

One-step derivation

Grammar

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

Example of one-step derivations

`Exp` $\rightarrow$ `Exp '*'Exp`	production (`Exp`,`Exp '*'Exp`) is used
`Exp '*'Exp` $\rightarrow$ `Num '*'Exp`	production (`Exp`,`Num`) is used
`Num '*'Exp` $\rightarrow$ `Num '*'Num`	production (`Exp`,`Num`) is used
`Num '*'Num` $\rightarrow$ `'0''*'Num`	production (`Num`,`'0'`) is used
`'0''*'Num` $\rightarrow$ `'0''*''1'`	production (`Num`,`'1'`) is used

Remarks

- there is no derivation from `'0''*''1'` (no production can be used)
- `'0''*''1'` is the string `"0*1"` which belongs to L_{Exp}

Definition of derivation

One-step derivation $\rightarrow$

One-step derivation for a grammar $G = (T, N, P)$

- it has shape $\alpha_1 B \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$
- $\alpha_1, \alpha_2 \in (T \cup N)^*$
- $(B, \gamma) \in P$ that is, (B, γ) is a production

Multi-step derivation $\rightarrow^+$

Transitive closure of $\rightarrow$:

- base case: if $\gamma_1 \rightarrow \gamma_2$, then $\gamma_1 \rightarrow^+ \gamma_2$
- inductive case: if $\gamma_1 \rightarrow \gamma_2$ and $\gamma_2 \rightarrow^+ \gamma_3$, then $\gamma_1 \rightarrow^+ \gamma_3$

Language generated

Language L_B generated from $G = (T, N, P)$ for non-terminal $B \in N$

- all strings of terminals that can be derived in one or more steps from B
- formally: $L_B = \{u \mid B \rightarrow^+ u\}$

Derivation tree (or parse tree)

Observation 1

- CF grammars are used to define languages and implement parsers
- Parsers should generate trees, but derivations are not tree!

Observation 2

- a derivation step is determined by
  1. the used production
  2. the specific non-terminal symbol which is replaced
- choice 2 does not influence the final string of terminals obtained from the derivation

Intuition

A derivation tree is a generalization of multi-step derivation such that

- the derived string contains only terminals
- non-terminal are replaced "in parallel"

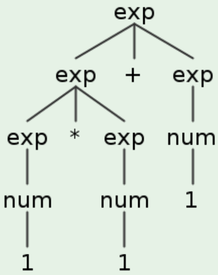
Examples of derivation trees (in ANTLR)

ANTLR Grammar

```
grammar SimpleExp;

exp :  num | exp '*' exp | exp '+' exp | '(' exp ')' ;
num : '0' | '1' ;
```

Derivation tree for `"1*1+1"`



Examples of derivation trees (in ANTLR)

ANTLR Grammar

```
grammar SimpleExp;

exp :  num | exp '*' exp | exp '+' exp | '(' exp ')' ;
num : '0' | '1' ;
```

Derivation tree for "(1+1)*1"