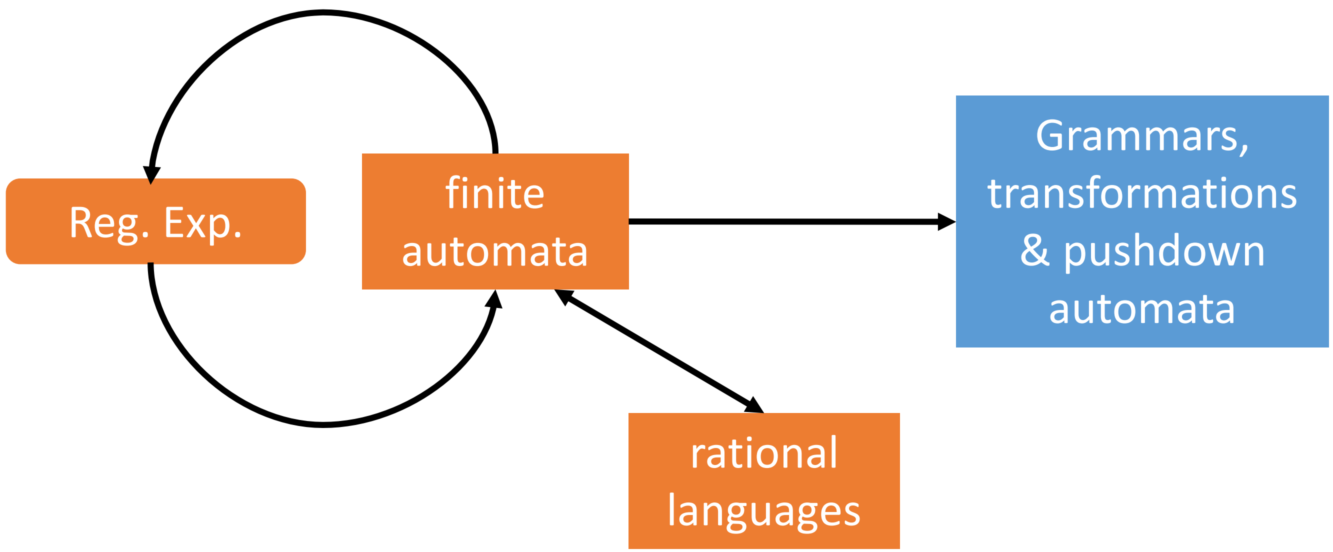
Grammaires régulières et algébriques



http://homepages.laas.fr/dalzilio/courses/ag/

2019—2020

Context-Free Grammars

grammaire algébrique [FR]

Context-Free Grammars (CFG)

A grammar $\equiv$ a set of rules for generating the elements of a language

We have already seen an example:

$$D \rightarrow \epsilon$$
$$D \rightarrow a\ D\ b$$
$$D \rightarrow D\ D$$

or equivalently

$$S \rightarrow \epsilon \mid \overbrace{a\ S\ b\ S}^{\text{word} \in (\mathrm{V}_t \cup V_{nt})^\star}$$

where we mix variables $S, D, X, \ldots$ (denoting sets) and symbols $a, b\ \epsilon, \ldots$ (and hence "words") in the right of production rules

Context-Free Grammars (CFG)

This is a tool for accepting/generating words, based on a notion of *derivations* ($\neq$ runs)

$$D \rightarrow \overline{a\ D\ b}$$

$a\ a\ b\ a\ b\ b$
is part of the language

$$D \rightarrow a\ D\ b$$

Context-Free Grammars (CFG)

This is a tool for accepting/generating words, based on a notion of *derivations* (≠ runs)

$$D \rightarrow \overline{a\ D\ b}$$
$$\rightarrow a\ \overline{D\ D}\ b$$

$a\ a\ b\ a\ b\ b$
is part of the language

$$D \rightarrow D\ D$$

Context-Free Grammars (CFG)

This is a tool for accepting/generating words, based on a notion of *derivations* (≠ runs)

$$
\begin{aligned}
{\color{#5B9BD5}D} &\to \overline{a\ D\ b} \\
&\to a\ \overline{{\color{#5B9BD5}D}\ D}\ b \\
&\to a\ \overline{a\ D\ b} D\ b
\end{aligned}
$$

$a\ a\ b\ a\ b\ b$ is part of the language

$$D \to a\ D\ b$$

Context-Free Grammars (CFG)

This is a tool for accepting/generating words, based on a notion of *derivations* (≠ runs)

$$
\begin{aligned}
D &\rightarrow \overline{a\ D\ b} \\
&\rightarrow a\ \overline{D\ D}\ b \\
&\rightarrow a\ \overline{a\ D\ b} D\ b \\
&\rightarrow a\ a\ \bar{\epsilon}\ b\ D\ b
\end{aligned}
$$

$a\ a\ b\ a\ b\ b$
is part of the language

$D \rightarrow \epsilon$

Context-Free Grammars (CFG)

This is a tool for accepting/generating words, based on a notion of *derivations* (≠ runs)

$$
\begin{aligned}
D &\rightarrow \overline{a\ D\ b} \\
&\rightarrow a\ \overline{D\ D}\ b \\
&\rightarrow a\ \overline{a\ D\ b} D\ b \\
&\rightarrow a\ a\ \bar{\epsilon}\ b\ D\ b \\
&\rightarrow a\ a\ b\ \overline{a\ D\ b}\ b
\end{aligned}
$$

$a\ a\ b\ a\ b\ b$ is part of the language

$$D \rightarrow a\ D\ b$$

Context-Free Grammars (CFG)

This is a tool for accepting/generating words, based on a notion of *derivations* ($\neq$ runs)

$$
\begin{aligned}
D &\to \overline{a\ D\ b} \\
&\to a\ \overline{D\ D}\ b \\
&\to a\ \overline{a\ D\ b} D\ b \\
&\to a\ a\ \bar{\epsilon}\ b\ D\ b \\
&\to a\ a\ b\ \overline{a\ D\ b}\ b \\
&\to a\ a\ b\ a\ b\ b
\end{aligned}
$$

$a\ a\ b\ a\ b\ b$ is part of the language

$$D \to \epsilon$$

Context-Free Grammars (CFG)

- Automaton $\Rightarrow$ define languages in an *operational* way (they are machines, or programs)

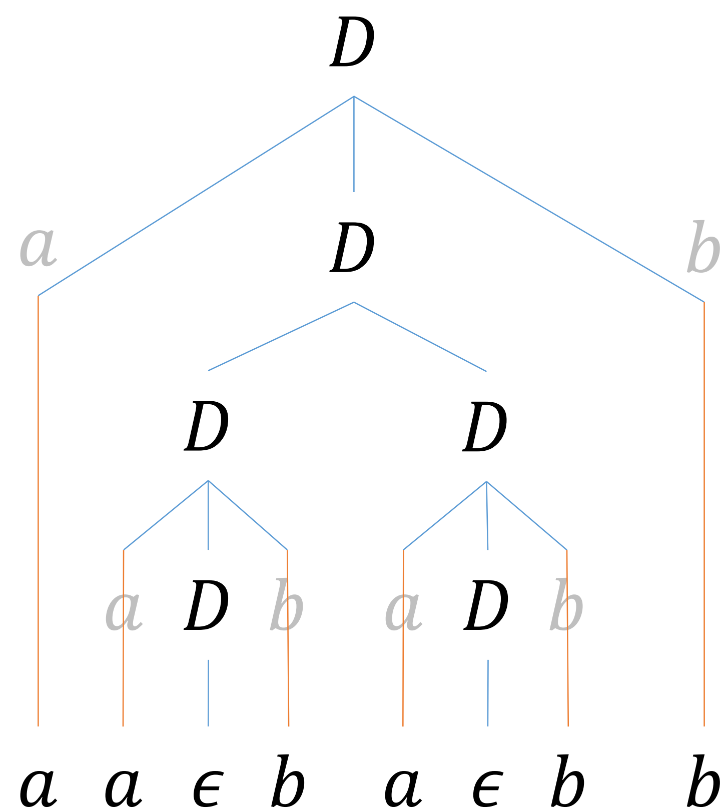
- Regex $\Rightarrow$ define languages in a *declarative* way (they are like functions or logical formulas)

- Grammars $\Rightarrow$ define languages in an *inductive* way

Context-Free Grammars (CFG)

This is best viewed as a *derivation tree*

$$
\begin{aligned}
D &\rightarrow \overline{a\ D\ b} \\
&\rightarrow a\ \overline{D\ D}\ b \\
&\rightarrow a\ \overline{a\ D\ b}D\ b \\
&\rightarrow a\ a\ \bar{\epsilon}\ b\ D\ b \\
&\rightarrow a\ a\ b\ \overline{a\ D\ b}\ b \\
&\rightarrow a\ a\ b\ a\ \bar{\epsilon}\ b\ b
\end{aligned}
$$



Context-Free Grammars (CFG)

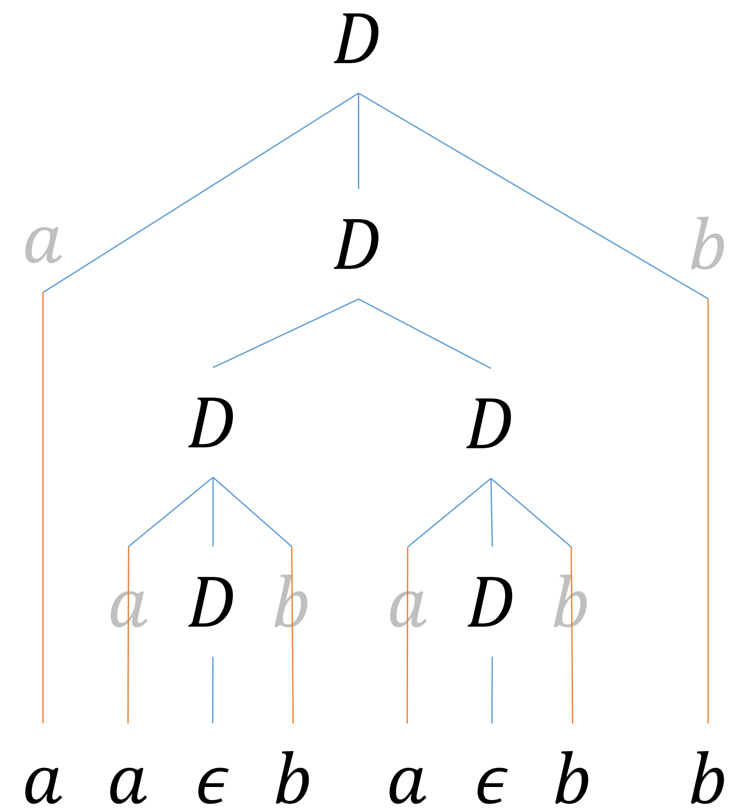
- a set of *production rules* of the form $X \rightarrow \alpha$
- with *non-terminal symbols* (variables in V_{nt})
- and *constants* (in V_t, sometimes denoted Σ)
- an *axiom*: distinguished symbol in V_t
- where $\alpha, \beta, \ldots$ are words in $(V_{nt} \cup V_t)^\star$

Non contextual grammars is the class obtained when we allow production rules of the form

$$\alpha \rightarrow \beta$$

Top-Down Derivations

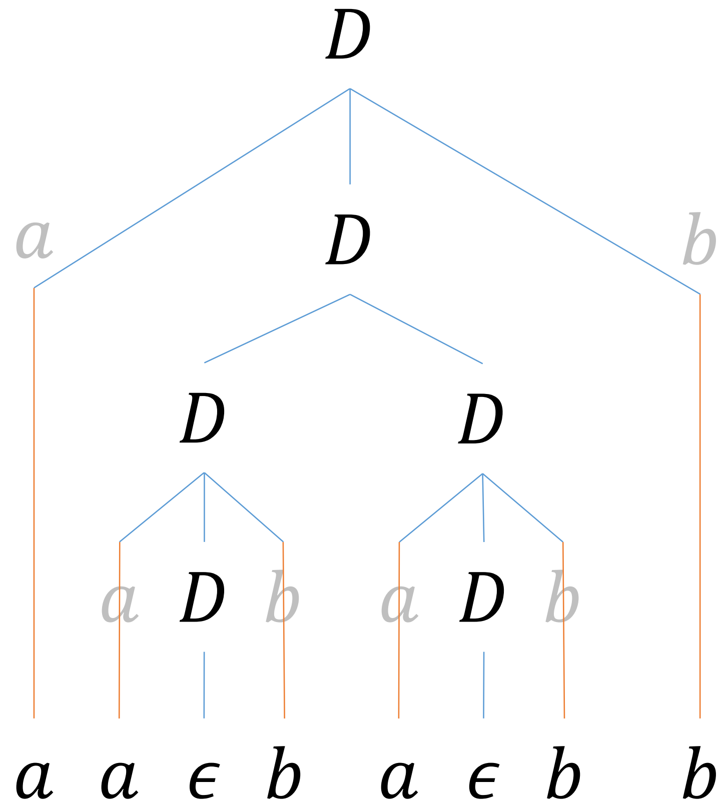
$$
\begin{aligned}
D &\rightarrow \overline{a\ D\ b} \\
&\rightarrow a\ \overline{D\ D}\ b \\
&\rightarrow a\ \overline{a\ D\ b}D\ b \\
&\rightarrow a\ a\ \bar{\epsilon}\ b\ D\ b \\
&\rightarrow a\ a\ b\ \overline{a\ D\ b}\ b \\
&\rightarrow a\ a\ b\ a\ \bar{\epsilon}\ b\ b
\end{aligned}
$$



Top-Down + Leftmost ≡ here we started from the axiom (the top) + we always decided to derive the leftmost non-terminal

Bottom-up Derivations

$$
\begin{aligned}
& a\ a\ b\ a\ b\ b \\
\Leftarrow\ & a\ a\ D\ b\ a\ b\ b \\
\Leftarrow\ & a\ D\ a\ b\ b \\
\Leftarrow\ & a\ D\ a\ D\ b\ b \\
\Leftarrow\ & a\ D\ D\ b \\
\Leftarrow\ & a\ D\ b \\
\Leftarrow\ & D
\end{aligned}
$$



Bottom-Up ≡ rules are considered in the order of a *reverse* rightmost derivation. In this case the tree is the same !

CFG and Automata

We already saw derivation rules before, but in a simpler setting:

$X_0 \to 0\,X_0$

$X_0 \to 1\,X_1$

$X_0 \to \Lambda$

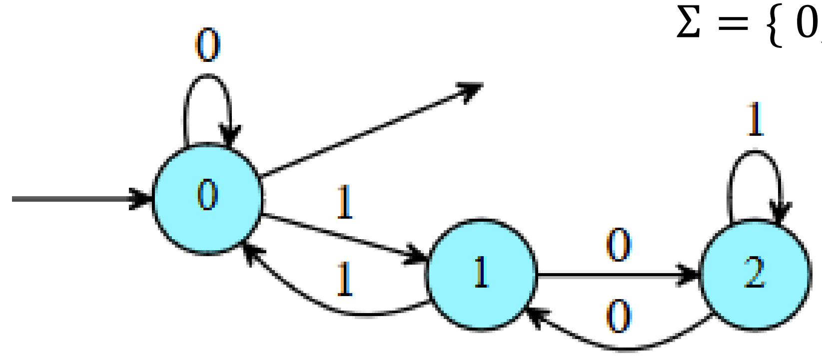
$X_1 \to 0\,X_2$

$X_1 \to 1\,X_0$

$X_2 \to 0\,X_1$

$X_2 \to 1\,X_2$

$\Sigma = \{\,0, 1\,\}$



$$\begin{array}{ccccccccc}
X_0 & — & X_1 & — & X_2 & — & X_1 & — & X_0 \\
| & & | & & | & & | & & | \\
1 & & 0 & & 0 & & 1 & & \epsilon
\end{array}$$

Questions

- Why (do we define CFG) ?
- What is the “power” of CFG ?
- Is it possible to check whether a word *is in* a CFG ?
- What is the “accepting device” (operational model) that corresponds to CFG ?
- Can we get rid of ambiguity ?
- Can we test if two grammars are $\equiv$?
- …

Parsing (and lexing)

answering the “why ?”

Parsing

An important application of CFG theory is in *parsing* programming language.

Human languages are a bit harder to parse … and too ambiguous !

(Extended) Backus-Naur form

A popular notation for CFG, used to define the syntax of programs and expressions in C.S.

```
⟨D⟩      ::= “1” | … | “9”
⟨DIGIT⟩ ::= “0” | ⟨D⟩
⟨INT⟩    ::= ⟨D⟩ ⟨DIGIT⟩*
```

extended means we can use $*$, $+$ and ?

Another Example of BNF

Arithmetic expressions:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathrm{n}$$

$$\begin{aligned} \langle \text{EXP} \rangle \quad ::= \; & \langle \text{EXP} \rangle \text{ '+' } \langle \text{EXP} \rangle \\ \mid \; & \langle \text{EXP} \rangle \text{ '*' } \langle \text{EXP} \rangle \\ \mid \; & \text{ '(' } \langle \text{EXP} \rangle \text{ ')' } \\ \mid \; & \langle \text{NUM} \rangle \end{aligned}$$

How should we "parse" expression `1+2*3` ?

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

A Real Example of BNF

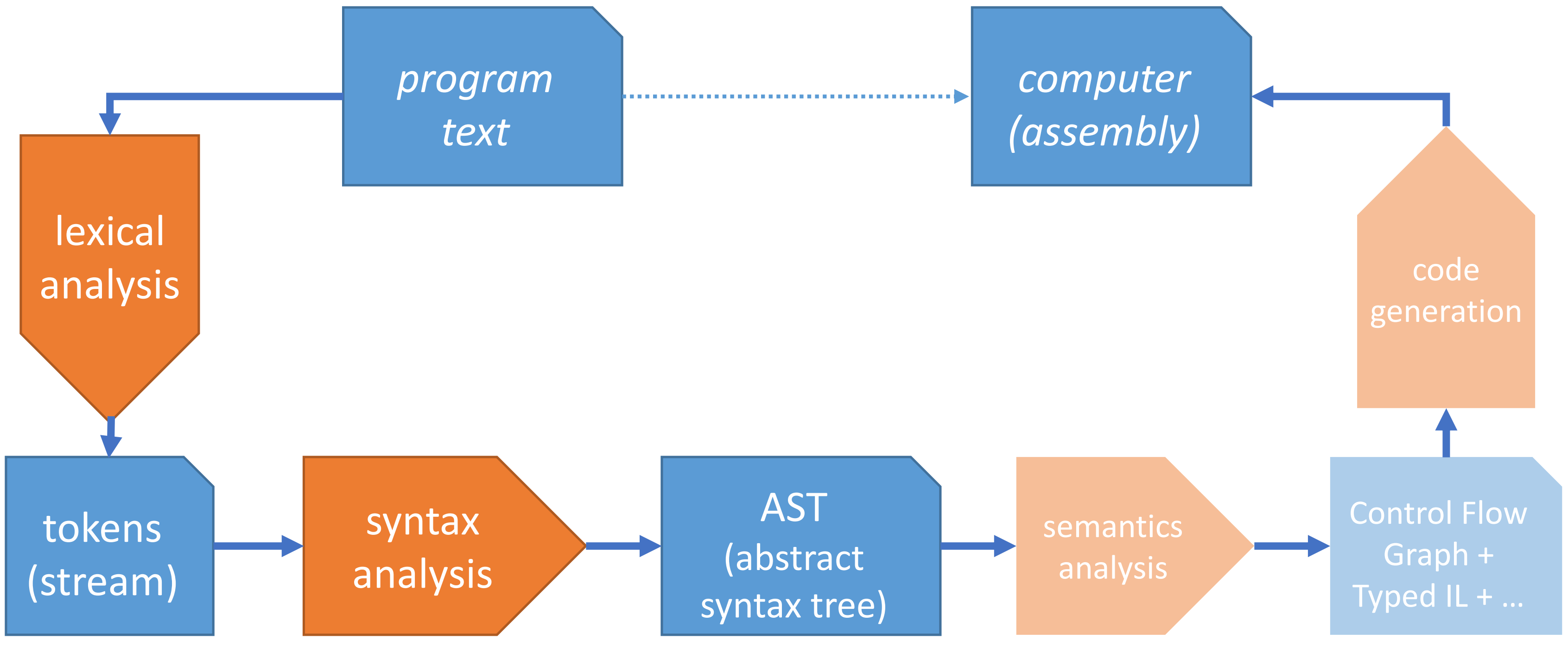
```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

The Go Programming Language Specification

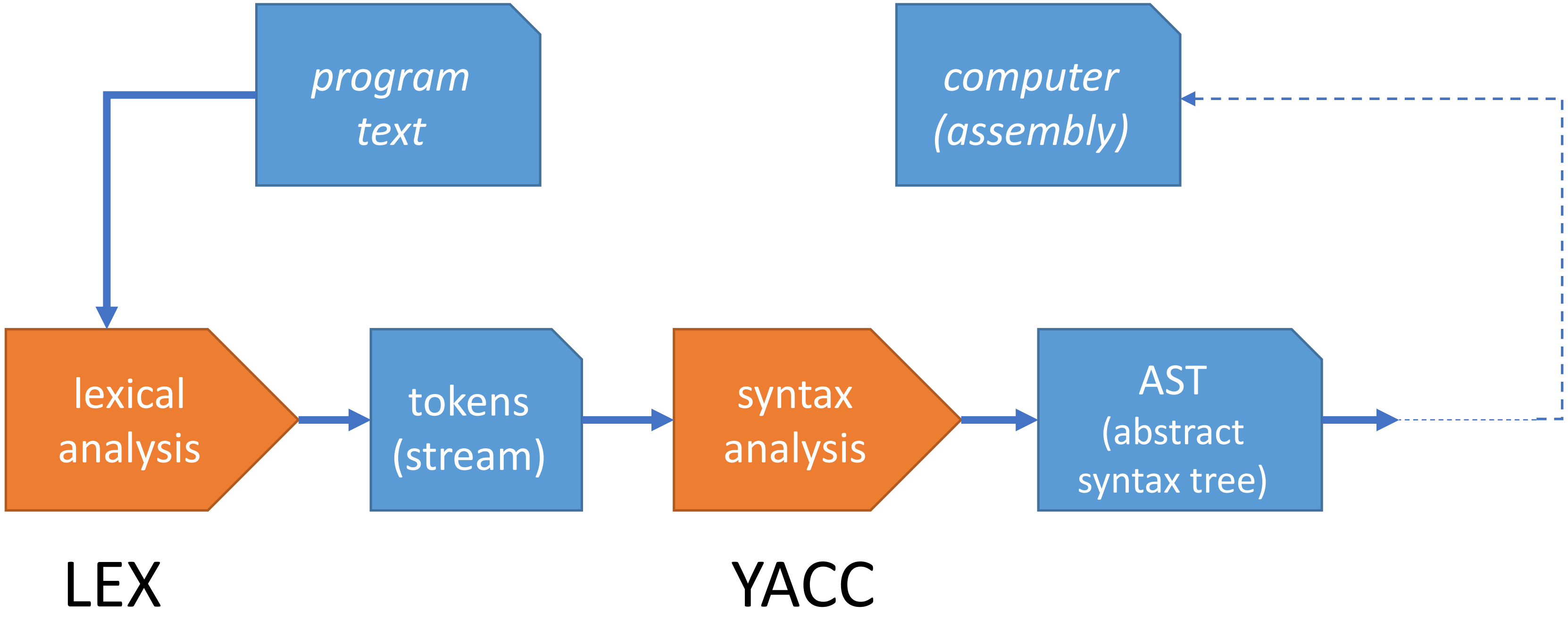
Parsing

An important application of CFG theory is in *parsing* programming language.



Lexing + Parsing

An important application of CFG theory is in *parsing* programming language.



Lexing + Parsing

lexical analysis, *lexing* or *tokenization* is the process of converting a sequence of *symbols* (characters) into a sequence of *tokens*.

Parsing, or *syntax analysis* is the process of analyzing a string of tokens conforming to the rules of a formal grammar.

Example: C-like function

```
func member(u []byte) bool {
     st := q0
     for i = 0; i < len(u); i++ {
      st = delta(st, u[i])
     }
     return isfinal(st)
}
```

Lexing

- keyword `func, for, if, len, return`
- identifiers `member, delta, u, …`
- separator `}, (, ;`
- operator `+, <, =, ++`
- literal `0, 1e999, "Hello, 世界"`
- comment `/* [\w]* */`

Example

```
func member(u []byte) bool {
    st := q0
    for i = 0; i < len(u); i++ {
      st = delta(st, u[i])
    }
    return isfinal(st)
}
```

func	member	(	u	[]byte	)	bool	{	…
kw	id	_	id	type	_	type	_	

Example

```go
func member(u []byte) bool {
    st := q0
    for i = 0; i < len(u); i++ {
        st = delta(st, u[i])
    }
    return isfinal(st)
}
```

func	member	(	u	[]byte	)	bool	{	…
kw	id	_	id	type	_	type	_	

FUNC	ID(“member”)	LPAR	ID(“u”)	…

Parsing

```
func member(u []byte) bool {
    st := q0
    for i = 0; i < len(u); i++ {
        st = delta(st, u[i])
    }
    return isfinal(st)
}
```

```
⟨FUNDECL⟩  ::= "func" ⟨ID⟩ "(" … ")" ⟨BLOCK⟩
⟨BLOCK⟩    ::= "{" ⟨EXPR⟩* "}"
⟨EXPR⟩     ::= "for" ⟨FCOND⟩ ⟨BLOCK⟩ | …
⟨ID⟩       ::= [a-zA-Z][\w\d]*
```

EBNF specification

Error detection

`FUNC`	`ID("member")`	`LPAR`	`ID("u")`	…
`ID("FUN")`	`ID("member")`	`LPAR`	`ID("u")`	…

It is generally during the syntactical analysis that we can spot errors.
Here a typo in the use of a keyword, because we have no rules of the form ⟨EXPR⟩ `::=` ⟨`ID`⟩ ⟨`ID`⟩.

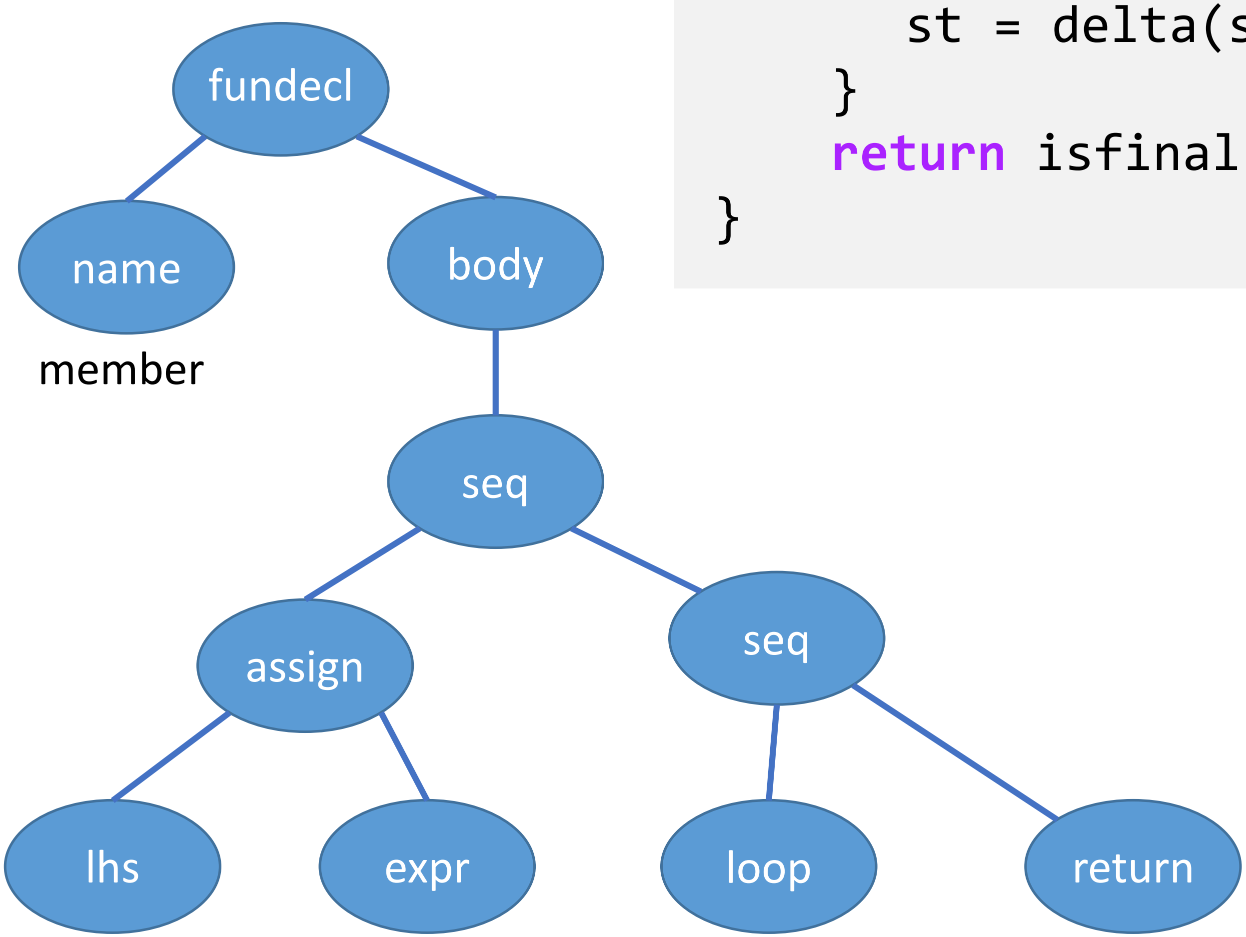
Lexing + Parsing

- Lexing ≡ generate a sequence of *lexemes*
  use of regex / finite state automata

- Parsing ≡ generate an Abstract Syntax Tree
  use Context-Free Grammars / an
  extension of automata
  (non-deterministic pushdown automata)

AST

```
func member(u []byte) bool {
     st := q0
     for i = 0; i < len(u); i++ {
        st = delta(st, u[i])
     }
     return isfinal(st)
}
```



Algebraic Grammars

formal definitions

Context-Free Grammar $\mathcal{G}$

$\mathcal{G}$ is a tuple $(\Sigma,\ \mathrm{V},\ P,\ S)$ where :

- Σ : alphabet, set of *terminal symbols* (also V_t)
- V : set of *non-terminal symbols* (also V_{nt})
- $S \in V$ axiom
- P : set of production rules $X \ \rightarrow \ \alpha$

Derivation

- We say that $\alpha \Rightarrow \beta$ in $\mathcal{G}$ if we have:
  $\alpha = \alpha_1 \, X \, \alpha_2$ and $X \ \rightarrow \gamma \ \in P$ and $\beta = \alpha_1 \, \gamma \, \alpha_2$

- We use $\Rightarrow^{\star}$ (or simply $\Rightarrow$) for the reflexive, transitive closure of $\Rightarrow$

Language

The language of a grammar $\mathcal{G}$ is the set of words

$$\mathcal{L}(\mathcal{G}) = \{\, w \in \Sigma^\star \mid S \Rightarrow^\star w \,\}$$

i.e. start with the axiom, ends only with symbols

Derivation tree of (Σ, V, P, S)

- a (finite, ordered) tree where nodes are decorated with symbols in $\Sigma \cup V$
- the root is decorated by S
- the leaves are in Σ
- a node "X“ has children $[\alpha_1, \ldots, \alpha_n]$ iff we have rule $X \rightarrow \ \alpha_1 \ \ldots \ \alpha_n \ \in P$ and $\alpha_i \in \Sigma \cup V$ for all $i \in 1..n$

Derivation : example

$\Sigma = \{ c, +, *, (,) \}$

$V = \{ E, F, T \}$

axiom: E

productions:

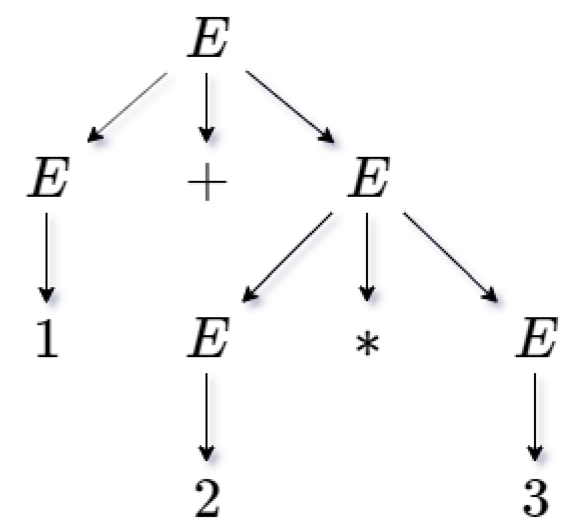
$$E \rightarrow F$$
$$E \rightarrow E + F$$
$$F \rightarrow T$$
$$F \rightarrow F * T$$
$$T \rightarrow c$$
$$T \rightarrow (E)$$

Ambiguity

```
⟨EXP⟩  ::= ⟨EXP⟩ + ⟨EXP⟩
         | ⟨EXP⟩ * ⟨EXP⟩
         | ( ⟨EXP⟩ )
         | ⟨NUM⟩
```

1 + 2 * 3 = 7

$$
\begin{aligned}
E &\to E + E \\
&\to 1 + E \\
&\to 1 + E * E \\
&\to 1 + 2 * E \\
&\to 1 + 2 * 3
\end{aligned}
$$

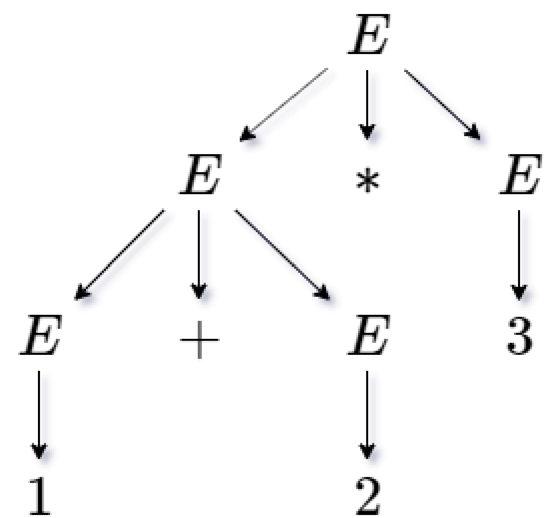


Ambiguity

$$
\begin{aligned}
\langle \texttt{EXP} \rangle \quad ::= \quad & \langle \texttt{EXP} \rangle \texttt{ + } \langle \texttt{EXP} \rangle \\
\mid \quad & \langle \texttt{EXP} \rangle \texttt{ * } \langle \texttt{EXP} \rangle \\
\mid \quad & \texttt{(} \langle \texttt{EXP} \rangle \texttt{)} \\
\mid \quad & \langle \texttt{NUM} \rangle
\end{aligned}
$$

`1 + 2 * 3 = 9`

$$
\begin{aligned}
E & \rightarrow E * E \\
& \rightarrow E * 3 \\
& \rightarrow E + E \ * 3 \\
& \rightarrow E + 2 \ * 3 \\
& \rightarrow 1 + 2 \ * 3
\end{aligned}
$$



Ambiguity

- A grammar is ambiguous if we can accept (at least) one word with (at least) two different derivations; two different parse trees

- // with NFA and non-determinism

- For the same language, we can sometimes have both ambiguous and non-ambiguous grammars; but there are also ambiguous languages

Ambiguity

There are ways to transform grammars:

$$
\begin{aligned}
\langle E \rangle &::= \langle F \rangle & &|\ \langle E \rangle\ \text{‘+’}\ \langle F \rangle \\
\langle F \rangle &::= \langle T \rangle & &|\ \langle F \rangle\ \text{‘*’}\ \langle T \rangle \\
\langle T \rangle &::= \langle NUM \rangle & &|\ \text{‘(‘}\ \langle E \rangle\ \text{‘)’}
\end{aligned}
$$

How should we “parse” expression 1 + 2 * 3

Answer: ~~(1 + 2) * 3~~ or 1 + (2 * 3)

Ambiguity

- A grammar $\mathcal{G}$ is ambiguous if we can find a word $w \in \mathcal{L}(\mathcal{G})$ that has two different derivation trees

  The set of all derivations for an ambiguous word is called a *parse forest*

- A (CFG) language is ambiguous if all the grammars that generate it are ambiguous

Ambiguity

- Leftmost-derivation: we always apply rules on the first non-terminal symbol (reading left-to-right)
- Rightmost-derivation: *what do you think !*

Theorem: A CFG is non-ambiguous iff there is a single leftmost ($\equiv$ rightmost) derivation

Ambiguity

Theorem: A CFG is non-ambiguous iff there is a single leftmost ($\equiv$ rightmost) derivation

Corollary: $\mathcal{G}$ non-ambiguous $\Rightarrow$ we can easily test if a word w is in $\mathcal{L}(\mathcal{G})$ (no backtracking)

In practice, there are two main sources of ambiguity

- we lack *priorities* between operators
- we lack *associativity rules* for the operators

Parsing CFG

Parser $\equiv$ an algorithm to test whether $w \in \mathcal{L}(\mathcal{G})$

There are three main classes of parsers:

1. Parser that works regardless of the CFG (no restrictions), e.g. *Earley parser*.
2. Top-down parser (for restricted class of non-ambiguous grammars), e.g. *LL parsers*.
3. Bottom-up parser (for a $\neq$ class of non-ambiguous grammars), e.g. *LR parsers*.

1 are mainly used for computational linguistics

2 + 3 are mainly used for (prog. lang.) compilers

Avoiding ambiguity

Ambiguity should be avoided:

⇒ we want to predict what non-terminal ($X \in V_{nt}$) to match, at any given moment, just by looking a few symbols ahead

⇒ we want to find *deterministic CFG*

Un fortunately, the problem of deciding whether a CFG is unambiguous is undecidable

⇒ we should restrict to sub-classes of grammars

⇒ examples are LL(k), LR(k), …

Questions

- Why ? ⇒ programming language !?
- Is it possible to check whether a word *is in* a CFG ?
- What is the "power" of CFG ?
- What is the "accepting device" (operational model) that corresponds to CFG ?
- Can we get rid of ambiguity ?
- Can we test if two grammars are ≡ ?
- …

Chomsky hierarchy

Chomsky-Schützenberger hierarchy

Noam Chomsky (1928—)

Marcel-Paul Schützenberger (1920—1996)

go check Wikipedia !

Grammar—Type 0

No constraints on the left/right part of production rules

Language: Recursively enumerable (R.E.)

Automaton: Turing machines

$$\alpha\, X\, \beta \;\rightarrow \gamma$$

Grammar—Type 1

soft constraint on the right part of production rules

Language: Context sensitive

Automaton: Linear Bounded Automaton

$$\alpha\, X\, \beta \;\rightarrow\; \alpha\, \gamma\, \beta$$

Example: Type1

$$
\begin{aligned}
S &\rightarrow a\ b\ c \mid a\ A\ b\ c \\
S &\rightarrow \Lambda \\
A\ b &\rightarrow b\ A \\
A\ c &\rightarrow B\ b\ c\ c \\
b\ B &\rightarrow B\ b \\
a\ B &\rightarrow a\ a\ A \mid a\ a
\end{aligned}
$$

This grammar generates words of the form: $a^n\ b^n\ c^n$

$$
\begin{aligned}
S &\rightarrow a\ \overline{A\ b}\ c \\
&\rightarrow a\ b\ \overline{A\ c} \\
&\rightarrow a\ \overline{b\ B}\ b\ c\ c \\
&\rightarrow \overline{a\ B}\ b\ b\ cc \\
&\rightarrow a\ a\ \overline{A\ b}\ b\ c\ c
\end{aligned}
$$

Grammar—Type 2

left part of production rules is in V_{nt}

Language: Context Free

Automaton: non-dét. Pushdown Automata

$$X \rightarrow \alpha$$

context-free implies that $\mathcal{L}(\alpha\ \beta) = \mathcal{L}(\alpha)\ \mathcal{L}(\beta)$

Example: Type2

$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow \Lambda$$
$$S \rightarrow a\, S\, a$$
$$S \rightarrow b\, S\, b$$

This grammar generates palindromes (words that read the same from left-right and right-left)

$$\begin{aligned} S &\rightarrow a\, \bar{S}\, a \\ &\rightarrow a\, b\, \bar{S}\, b\, a \\ &\rightarrow a\, b\, b\, \bar{S}\, b\, b\, a \\ &\rightarrow a\, b\, b\, b\, b\, a \end{aligned}$$

Grammar—Type 3

No constraints on the left/right part of production rules

Language: Regular

Automaton: DFA

$$X \to a\, Y$$

Type 3 grammars ≡ Rational Languages

Sketch of the proof: Type3 ⊆ REG

Build an automaton $\mathcal{A}$ where

- states: $Q = V_{nt} \cup \{ q_F \}$
- initial state: $q_I = S$ (axiom)
- If $X \rightarrow a\,Y$ then $\delta(X, a) = Y$
- If $X \rightarrow a$ then $\delta(X, a) = q_F$
- If $X \rightarrow \epsilon$ then $\delta(X, \epsilon) = q_F$
- final states: $\{ q_F \}$

We can build a run in $\mathcal{A}$ from any derivation

Sketch of the proof: REG ⊆ Type3

Build a grammar from $\mathcal{A}$ where

- $V_{nt} = Q$ and $V_t = \Sigma$
- axiom is q_I
- If $\delta(X, a) = Y$ then add $X \rightarrow a\, Y$
- If $\delta(X, a) = Y$ and $Y \in F$ then add $X \rightarrow a$

We can build a run in $\mathcal{A}$ from any derivation

Example

$S \rightarrow a\,S \mid b\,E \mid \Lambda$

$E \rightarrow b\,E \mid \Lambda$

This grammar generates words of the form $a^n b^m$.

What about the Type-2 grammar:

$S \rightarrow A\,B$

$A \rightarrow a\,A \quad | \, B$

$B \rightarrow B\,b\,B \mid b\,b\,B \mid \Lambda$

Rewriting grammars

$\equiv$ operations that "preserve" the language of a grammar $\mathcal{G}$

Notations

Without loss of generality, we can always write production rules of the form $X \rightarrow \alpha \mid \beta$ as "syntactic sugar" for the two rules:

$X \rightarrow \alpha$ and $X \rightarrow \beta$

Substitution

Assume all the production rules for X in $\mathcal{G}$ are

$X \ \rightarrow \alpha_1, \ldots, X \ \rightarrow \alpha_k$

Then we can replace a production rule $Y \ \rightarrow \beta\, X\, \gamma$, in $\mathcal{G}$, with the k rules: $Y \rightarrow \beta\, \alpha_1\, \gamma\ , \ldots, Y \rightarrow \beta\, \alpha_k\, \gamma$

$$S \ \rightarrow a\, X \ | \ \ c$$
$$X \ \rightarrow Y\, a \,|\, \Lambda$$
$$Y \ \rightarrow \alpha$$

$$\Rrightarrow$$

$$S \ \rightarrow a\, Y\, a$$
$$S \ \rightarrow a$$
$$S \ \rightarrow \Lambda$$
$$\ldots$$

Simplification

Assume it is not possible* to find a sequence of reductions (from the axiom S) such that $S \Rightarrow^\star \ \beta\ X\ \gamma$

Then we can safely omit all the productions of the form $X \ \rightarrow \alpha$ from $\mathcal{G}$

$$\begin{array}{lll} S \ \rightarrow a\ Y \mid \Lambda & & S \ \rightarrow a\ Y \mid \Lambda \\ Y \ \rightarrow a\ S & \Rrightarrow & Y \ \rightarrow a\ S \\ X \ \rightarrow Y\ Y & & \cancel{X \ \rightarrow Y\ Y} \end{array}$$

[*]: think reachability in a graph

Factorization

We can introduce new variables. For instance to factorize common sub-terms

$$\begin{aligned} X &\to \alpha\, A \\ X &\to \alpha\, B \end{aligned} \qquad \Rrightarrow \qquad \begin{aligned} X &\to \alpha\, Y \\ Y &\to A \\ Y &\to B \end{aligned}$$

Recursion elimination

Left recursion often poses problems for parsers, e.g. it leads them into infinite recursion. Recursion can be direct (e.g. $X \rightarrow X\ X$) or indirect.

Example: $X \rightarrow Y$ and $Y \rightarrow X\ a$

Recursion elimination

In the general case we have (where $\alpha_i \neq \Lambda$ and β_j does not start with X)

$$X \to X\,\alpha_1 \mid \ldots \mid X\,\alpha_k \mid \beta_1 \mid \ldots \mid \beta_n$$

Idea: introduce a "fresh" variable Y with the rules

$$X \to \beta_1\,Y \mid \ldots \mid \beta_n\,Y$$
$$Y \to \alpha_1\,Y \mid \ldots \mid \alpha_k\,Y \mid \Lambda$$

Left Recursion: example

Take the classical example of integer expressions

$$E \to E + E \mid n$$

$$\alpha_1 = +E$$
$$\beta_1 = n$$

We obtain:

$$E \to n\ Y$$
$$Y \to +E\ Y \mid \Lambda$$

$$E \to \beta_1\ Y$$
$$Y \to \alpha_1\ Y \mid \Lambda$$

Questions

- Why ? ⇒ programming language !?
- Is it possible to check whether a word *is in* a CFG ?
- Can we build a “deterministic” parser ?
- What is the “accepting device” (operational model) that corresponds to CFG ?
- Can we test if two grammars are ≡ ?
- …

LL grammars

recursive descent parsers

LL parsers

LL grammars are those than can be parsed using a LL parser ⇒ parses by scanning the input from **Left** to right and building a **Leftmost** derivation

They can be parsed by a (top-down) *recursive descent parser* ; LL(k) grammars correspond to parser that take their decision based on a look-ahead of k symbols (and without backtracking)

We look at an example of LL(1) grammars next

LL parsers: example

1. $S \rightarrow a\, S\, b\, T$
2. $S \rightarrow c\, T$
3. $S \rightarrow d$
4. $T \rightarrow a\, T$
5. $T \rightarrow b\, S$
6. $T \rightarrow c$

$$w = a\ c\ c\ b\ b\ a\ d\ b\ c$$

$$S \rightarrow a \circ S\, b\, T \qquad (1)$$

LL parsers

1. $S \rightarrow a\, S\, b\, T$
2. $S \rightarrow c\, T$
3. $S \rightarrow d$
4. $T \rightarrow a\, T$
5. $T \rightarrow b\, S$
6. $T \rightarrow c$

$w = a\ c\ c\ b\ b\ a\ d\ b\ c$

$$
\begin{array}{llr}
S & \rightarrow a \circ S\ b\ T & (1) \\
 & \rightarrow a\ c \circ T\ b\ T & (2) \\
 & \rightarrow a\ c\ c \circ b\ T & (6) \\
 & \rightarrow a\ c\ c\ \ b \circ T & \\
 & \rightarrow a\ c\ c\ \ b\ b \circ S & (5) \\
 & \rightarrow a\ c\ c\ \ b\ b\ a \circ S\ b\ T & (1) \\
 & \rightarrow a\ c\ c\ \ b\ b\ a\ d \circ b\ T & (3) \\
 & \rightarrow a\ c\ c\ \ b\ b\ a\ d\ b \circ T & \\
 & \rightarrow a\ c\ c\ \ b\ b\ a\ d\ b\ c \circ \epsilon & (6)
\end{array}
$$

Questions

- What is a suitable *accepting device* for this example ?
- How can I check that my grammar is LL(1) ?
- If it is not, is there a way to repair it ?

LL Parser

At each step we have a derivation of the form $u \circ \alpha$ where u is a prefix of w of length i ($u = w[:i]$)

$\Rightarrow$ we match the suffix $w[i:]$ with α

We decide what rule to match by looking at the next symbol (say $w[i+1] = a$)

$\Rightarrow$ the choice should be unique, depending only on the top symbol ($w[i]$) and the start of α

$\Rightarrow$ we could encode this " function" in a table

LL Parser

At each step, we try to match a suffix, $w[i:]$, with a pattern α

(SHIFT) $\alpha = b\,\gamma$ and $w[i] = b$

we try to match word $w[i+1:]$ with pattern γ

(REDUCE) $\alpha = X\,\gamma$

we need to match symbol a with $X \rightarrow \beta$

we continue with $w[i:]$ and the pattern $\beta\,\gamma$

(STOP) we matched the whole word and $\alpha = \epsilon$,

or when we have no rules to match (ERROR)

LL Parser: amelioration

To make sure we match the axiom, S, we add a new symbol, $\$$, and a new top-level axiom rule $S' \to S\ \$$

$\Rightarrow$ the initial pattern is $S\ \$$

Possible cases for errors are:

- we “shift” a bad symbol: $\alpha = b\ \gamma$ and $w[i] \neq b$
- we reach the end of the input ($\$$) and $\alpha \neq \$$
- we reach the end of the pattern $\alpha = \$$ and $w[i] \neq \$$

Parsing Table

$$\begin{array}{llllll} S' & \rightarrow & S\,\$ & & & \\ S & \rightarrow & a\,S\,b\,T & \mid c\,T & \mid d \\ T & \rightarrow & a\,T & \mid b\,S & \mid c \end{array}$$

To match a symbol a, and a non-terminal, X, to a rule, $X \rightarrow \beta$, we assume that we computed a parsing table

	a	b	c	d
S	$a\,S\,b\,T$		$c\,T$	d
T	$a\,T$	$b\,S$	c	

	a	b	c	d
S	$a\,S\,b\,T$		$c\,T$	d
T	$a\,T$	$b\,S$	c	

$$S' \to S\,\$$$
$$S \to a\,S\,b\,T \mid c\,T \mid d$$
$$T \to a\,T \mid b\,S \mid c$$

$a\ c\ c\ b\ a\ c\ \$$	parse successful
$a\ a\ b\ c\ d\ d\ d\ \$$	illegal input
$a\ a\ d\ c\ a\ a\ c\ c\ \$$	illegal input

Recursive descent parser

```
func member(u []byte) bool {
    st, i := stack("S", "$"), 0
    for {
      if i == len(u) || len(stack) == 0 {
        return false
      }
      α := stack.pop()
      switch {
        case α == "$" && u[i] == "$":
            return true
        case α == u[i]: i++
        case α.(nterm):
            stack := stack.push(reduce(α, u[i]))
        default : return false
} } }
```

LL grammars

building a parsing table

Questions

- What is a suitable *accepting device* for this example ?

→ How can I check that my grammar is LL(1) ?

- If it is not, is there a way to repair it ?

Building the parsing table

A grammar is LL

$\Longleftrightarrow$ we can build a LL parser from it

$\Longleftrightarrow$ we can build a (LL) parsing table

Next we show how to build this table by computing three different relations: FIRST, NULL and FOLLOW

LL Parser: FIRST

To build a table, we need to know: what symbols can "appear first", at the beginning of a non-terminal X and to which production $X \rightarrow \alpha$ it belongs

E.g. we want to match $a\ w$ with pattern $X\ \gamma$ and we have a rule $X \rightarrow a\ Y$

Also, we should not have $X \rightarrow a\ Y$ and $X \rightarrow a\ Z$

LL Parser: NULL

Therefore we should also know when a non-terminal X is *nullable*, that is $X \Rightarrow^\star \epsilon$

E.g. we want to match a with pattern $X\ \gamma$ and we are in a situation where $X \Rightarrow^\star \epsilon$

Also, we should not match symbol a with Z when $Z \rightarrow X\ Y$ with $X \rightarrow a\ \gamma \mid \Lambda$ and $Y \rightarrow a\ \gamma'$

LL Parser: FOLLOW

Meaning, we should know the symbols that can follow a non-terminal X.

E.g. when we want to match symbol a with pattern $X\ Y$, a possible solution is that $X \Rightarrow^{\star} \epsilon$ and $Y \Rightarrow^{\star}\ a\ \gamma$

NULL, FIRST and FOLLOW

We have FIRST(α) $= \{ b \in \Sigma \mid \alpha \Rightarrow^\star b\,\gamma \}$

We say that null(α) when $\alpha \Rightarrow^\star \epsilon$

this is decidable

We have FOLLOW(X) $= \{ a \in \Sigma \mid S \Rightarrow^\star \beta\, X\, a\, \gamma \}$

Ambiguity $\Rightarrow$ we should not find two rules
$X \rightarrow \alpha$ and $X \rightarrow \beta$ such that
FIRST$(\alpha) \cap$ FIRST$(\beta) \neq \emptyset$

a FIRST-FIRST conflict

Ambiguity revisited

Actually, we can prove that the grammar is LL(1) when, for every non-terminal X with productions $X \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$, we have that:

For every pair $X \rightarrow \alpha$ and $X \rightarrow \beta$ we have FIRST$(\alpha) \cap$ FIRST$(\beta) = \emptyset$

no FIRST-FIRST conflicts

if NULL(X) then FIRST$(\alpha_i) \cap$ FOLLOW$(X) = \emptyset$

no FIRST-FOLLOW conflicts

LL Parser: FIRST

We have $\mathrm{FIRST}(\alpha) = \{\, b \in \Sigma \mid \alpha \Rightarrow^{\star}\ b\,\gamma \,\}$

$$\mathrm{FIRST}(\epsilon) = \emptyset$$

$$\mathrm{FIRST}(a) = \{\, a \,\}$$

$$\mathrm{FIRST}(\alpha_1\ \ldots \alpha_n) = \ \bigcup_{i \in 1..n} \{\, \mathrm{FIRST}(\alpha_i) \mid \mathrm{null}\big(\alpha_j\big), j < i \,\}$$

Equivalently: FIRST is the smallest relation such that $X\ \to Y_1\ \ldots Y_n\, Z\, \beta$ implies $\mathrm{FIRST}(Z) \subseteq \mathrm{FIRST}(X)$ when $Y_1, \ldots, Y_n$ are all nullable.

LL Parser: FOLLOW

We have FOLLOW$(X) = \{ a \in \Sigma \mid S \Rightarrow^{\star} \beta\, X\, a\, \gamma \}$

(and we assume FOLLOW$(S) \supseteq \{ \$ \}$)

FOLLOW is the smallest relation such that:

$A \to \alpha\, X\, Y_1\, \ldots Y_n\, Z\, \beta$ implies
FIRST$(Z) \subseteq$ FOLLOW(X) when $Y_1, \ldots, Y_n$ nullable.

$A \to \alpha\, X\, Y_1\, \ldots Y_n\, Z$ implies
FOLLOW$(A) \subseteq$ FOLLOW(X) when $Y_1, \ldots, Y_n$ nullable.

Symboles Directeurs (SD)

Dans les notations utilisées à l'ENSEEIHT, on fait usage de la notion de *symbole directeur* pour une production $X \ \rightarrow \alpha$.

$\mathrm{SD}(X \ \rightarrow \alpha) = \mathrm{FIRST}(\alpha)$ si $\alpha \neq \Lambda$

$\mathrm{SD}(X \ \rightarrow \Lambda) = \mathrm{FOLLOW}(X)$

conflits LL $\equiv$ le même symbole dans deux règles

$$\mathrm{SD}(X \ \rightarrow \alpha) \text{ et } \mathrm{SD}(X \ \rightarrow \beta)$$

Symboles Directeurs (SD)

Avantage 1: un critère unique pour reconnaître l'ambiguité d'une grammaire.

Avantage 2: si on veut matcher $w[i:]$ (avec symbole de tête b) contre le non-terminal X; il suffit de choisir l'unique production $X \to \alpha$ telle que $b \in \mathrm{SD}(X \to \alpha)$

		b	c	…
X		α	$\emptyset$	…
…		…	…	…

$b \in \mathrm{SD}(X \to \alpha)$

Example

$$S \rightarrow A\ B\ S \mid d$$
$$A \rightarrow B \mid a$$
$$B \rightarrow c \mid \Lambda$$

	NULL	FIRST	FOLLOW
S	no	$\{a, c, d\}$	$\{\$\}$
A	yes	$\{a, c\}$	$\{a, c, d\}$
B	yes	$\{c\}$	$\{a, c, d\}$

Example

$$S \rightarrow A\ B\ S \mid d$$
$$A \rightarrow B \mid a$$
$$B \rightarrow c \mid \Lambda$$

SD$(S \rightarrow A\ B\ S) \cap$
SD$(S \rightarrow d) = \{\, d \,\}$

FIRST-FIRST conflict

	NULL	FIRST	FOLLOW
S	no	$\{\, a, c, d \,\}$	$\{\, \$ \,\}$
A	yes	$\{\, a, c \,\}$	$\{\, a, c, d \,\}$
B	yes	$\{\, c \,\}$	$\{\, a, c, d \,\}$

SD$(B \rightarrow c) \cap$
SD$(B \rightarrow \Lambda) = \{\, c \,\}$

FIRST-FOLLOW conflict

Example

$$S \rightarrow i\, E\, t\, S\, e\, S \mid c$$

$$E \rightarrow b$$

	NULL	FIRST	FOLLOW
S	no	$\{ i, c \}$	$\{ e, \$ \}$
E	no	$\{ b \}$	$\{ t \}$

Example

$$S' \to \$$$

$$S \to i\ E\ t\ S\ e\ S \mid c$$

$$E \to b$$

$\mathrm{SD}(S \to i\ E\ \ldots) = \{\, i \,\}$

$\mathrm{SD}(S \to c) = \{\, c \,\}$

$\mathrm{SD}(E \to b) = \{\, b \,\}$

	NULL	FIRST	FOLLOW
S	no	$\{\, i, c \,\}$	$\{\, e, \$ \,\}$
E	no	$\{\, b \,\}$	$\{\, t \,\}$

Example

$S' \to \$$

$S \to i\ E\ t\ S\ e\ S \mid c$

$E \to b$

$\mathrm{SD}(S \to i\ E\ \ldots) = \{\ i\ \}$

$\mathrm{SD}(S \to c) = \{\ c\ \}$

$\mathrm{SD}(E \to b) = \{\ b\ \}$

$w = i\ b\ t\ c\ e\ c\ \$$

$\gamma_0 = S\ \$$

	i	t	e	c	b
S	$i\ E\ t\ S\ e\ S$			c	
E					b

Example

$$X \to Y\ c \mid a$$
$$Y \to b\ Z \mid \Lambda$$
$$Z \to \Lambda$$

	NULL	FIRST	FOLLOW
X			
Y			
Z			