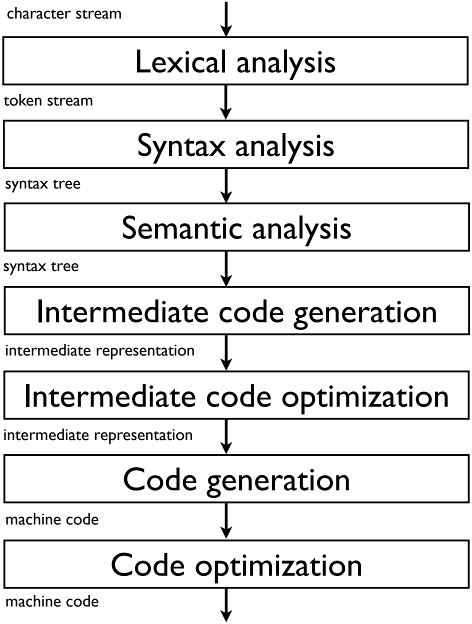
Part 3

Syntax analysis

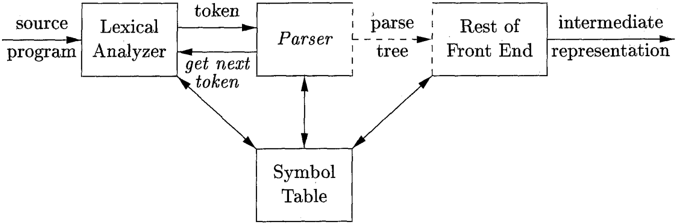
Outline

3.1 Introduction

Structure of a compiler



Syntax analysis



- Goals of the syntax analysis
  - Recombine the tokens provided by the lexical analysis into a structure (called a *syntax tree*)
  - Reject invalid texts by reporting *syntax errors*.
- Like lexical analysis, syntax analysis is based on
  - the definition of valid programs based on some formal languages,
  - the derivation of an algorithm to detect valid words (programs) from this language
- Formal language: context-free grammars
- Two main algorithm families: Top-down parsing and Bottom-up parsing

Example

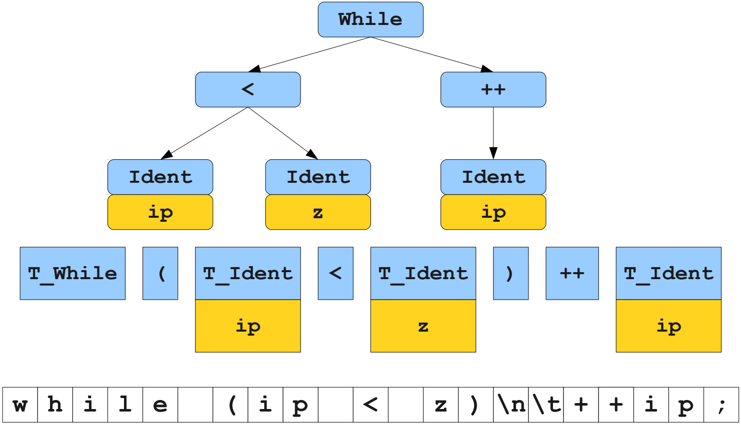
T_While	(	T_Ident	<	T_Ident	)	++	T_Ident
		ip		z			ip

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Example




```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

3.2
Context-free grammars

Reminder: grammar

- A grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where:
  - V is an alphabet,
  - $\Sigma \subseteq V$ is the set of terminal symbols
    ($V - \Sigma$ is the set of nonterminal symbols),
  - $R \subseteq (V^+ \times V^*)$ is a finite set of production rules
  - $S \in V - \Sigma$ is the start symbol.
- Notations:
  - Nonterminal symbols are represented by uppercase letters: A,B,...
  - Terminal symbols are represented by lowercase letters: a,b,...
  - Start symbol written as S
  - Empty word: ε
  - A rule $(\alpha, \beta) \in R$: $\alpha \to \beta$
  - Rule combination: $A \to \alpha \mid \beta$

Reminder: grammar

- Example: $\Sigma = \{a, b, c\}$, $V - \Sigma = \{S, R\}$, $R =$

$$
\begin{array}{lcl}
S & \rightarrow & R \\
S & \rightarrow & aSc \\
R & \rightarrow & \varepsilon \\
R & \rightarrow & RbR
\end{array}
$$

Reminder: derivation and language

Definitions:

- v can be *derived in one step* from u by G (noted $u \Rightarrow v$) iff $u = xu'y$, $v = xv'y$, and $u' \rightarrow v'$
- v can be *derived in several steps* from u by G (noted $u \stackrel{*}{\Rightarrow} v$) iff $\exists k \geq 0$ and $v_0 \ldots v_k \in V^+$ such that $u = v_0$, $v = v_k$, $v_i \Rightarrow v_{i+1}$ for $0 \leq i < k$
- The *language generated by a grammar* G is the set of words that can be derived from the start symbol:

$$L = \{w \in \Sigma^* | S \stackrel{*}{\Rightarrow} w\}$$

Example: derivation of *aabcc* from the previous grammar

$$\underline{S} \Rightarrow a\underline{S}c \Rightarrow aa\underline{S}cc \Rightarrow aa\underline{R}cc \Rightarrow aa\underline{R}bRcc \Rightarrow aab\underline{R}cc \Rightarrow aabcc$$

Reminder: type of grammars

Chomsky's grammar hierarchy:

- Type 0: free or unrestricted grammars
- Type 1: context sensitive grammars
  - productions of the form $\alpha X \beta \rightarrow \alpha\gamma\beta$, where α, β, γ are arbitrary strings of symbols in V, with γ non-null, and X a single nonterminal
- Type 2: context-free grammars (CFG)
  - productions of the form $X \rightarrow \alpha$ where α is an arbitrary string of symbols in V, and X a single nonterminal.
- Type 3: regular grammars
  - Productions of the form $X \rightarrow a$, $X \rightarrow aY$ or $X \rightarrow \varepsilon$ where X and Y are nonterminals and a is a terminal (equivalent to regular expressions and finite state automata)

Context-free grammars

- Regular languages are too limited for representing programming languages.
- Examples of languages not representable by a regular expression:
  - $L = \{a^n b^n \mid n \geq 0\}$
  - Balanced parentheses
    $L = \{\varepsilon, (), (()), ()(), ((())), (())() \ldots\}$
  - Scheme programs
    $L = \{1, 2, 3, \ldots, (lambda(x)(+\ x\ 1))\}$

- Context-free grammars are typically used for describing programming language syntaxes.
  - They are sufficient for most languages
  - They lead to efficient parsing algorithms

Context-free grammars for programming languages

- Terminals of the grammars are typically the tokens derived by the lexical analysis (in **bold** in rules)
- Divide the language into several syntactic categories (sub-languages)
- Common syntactic categories:
  - Expressions: calculation of values
  - Statements: express actions that occur in a particular sequence
  - Declarations: express properties of names used in other parts of the program

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp\ \mathbf{+}\ Exp \\
Exp & \rightarrow & Exp\ \mathbf{-}\ Exp \\
Exp & \rightarrow & Exp\ \mathbf{*}\ Exp \\
Exp & \rightarrow & Exp\ \mathbf{/}\ Exp \\
Exp & \rightarrow & \mathbf{num} \\
Exp & \rightarrow & \mathbf{id} \\
Exp & \rightarrow & \mathbf{(}\ Exp\ \mathbf{)}
\end{array}
$$

$$
\begin{array}{lcl}
Stat & \rightarrow & \mathbf{id}\ \mathbf{:=}\ Exp \\
Stat & \rightarrow & Stat\ \mathbf{;}\ Stat \\
Stat & \rightarrow & \mathbf{if}\ Exp\ \mathbf{then}\ Stat\ \mathbf{else}\ Stat \\
Stat & \rightarrow & \mathbf{if}\ Exp\ \mathbf{then}\ Stat
\end{array}
$$

Derivation for context-free grammar

- Like for a general grammar
- Because there is only one nonterminal in the LHS of each rule, their order of application does not matter (but the chosen rules do)
- Two particular derivations:
  - left-most: always expand first the left-most nonterminal (important for parsing)
  - right-most: always expand first the right-most nonterminal (canonical derivation)
- Examples:

$$S \rightarrow aTb \mid c$$
$$T \rightarrow cSS \mid S$$

$$w = accacbb$$

Left-most derivation:

$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow accSb \Rightarrow accaTbb \Rightarrow accaSbb \Rightarrow accacbb$

Right-most derivation:

$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow acSaTbb \Rightarrow acSaSbb \Rightarrow acSacbb \Rightarrow accacbb$

Parse tree

- A parse tree abstracts the order of application of the rules
  - Each interior node represents the application of a production
  - For a rule $A \rightarrow X_1 X_2 \ldots X_k$, the interior node is labeled by A and the children from left to right by $X_1, X_2, \ldots, X_k$.
  - Leaves are labeled by nonterminals or terminals and, read from left to right, represent a string generated by the grammar

- A derivation encodes how to produce the input
- A parse tree encodes the structure of the input

- Syntax analysis = recovering the parse tree from the tokens

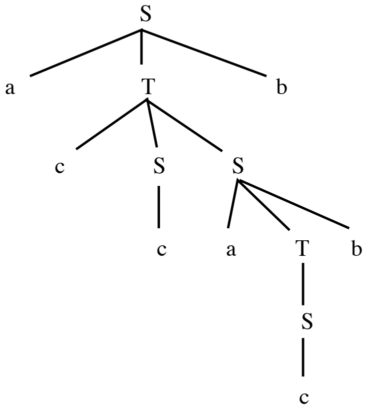
Parse trees

$$
\begin{array}{lcl}
S & \rightarrow & aTb \mid c \\
T & \rightarrow & cSS \mid S
\end{array}
$$

$$w = accacbb$$

Left-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow accSb \Rightarrow accaTbb \Rightarrow accaSbb \Rightarrow accacbb$

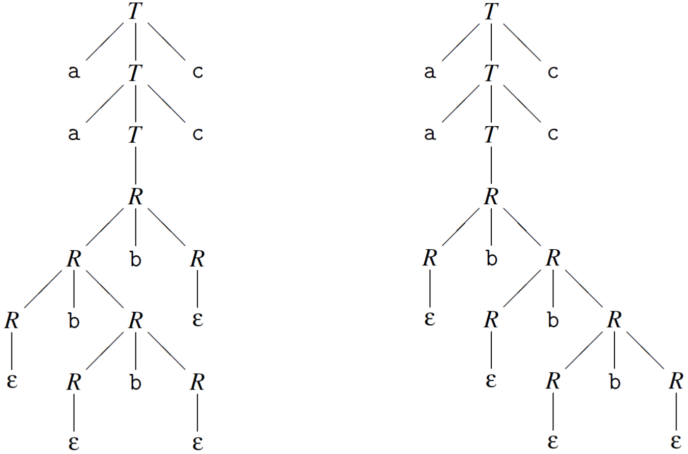
Right-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow acSaTbb \Rightarrow acSaSbb \Rightarrow acSacbb \Rightarrow accacbb$



Parse tree

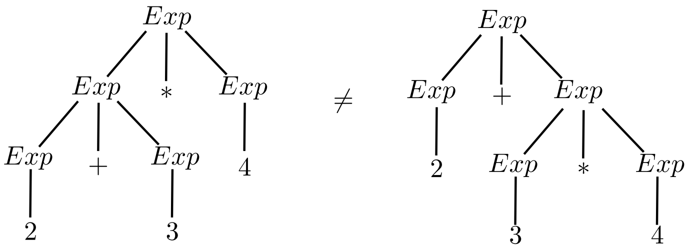
$$T \rightarrow R \mid aTc$$
$$R \rightarrow RbR \mid \varepsilon$$

$$w = aabbbcc$$



Ambiguity

- The order of derivation does not matter but the chosen production rules do
- Definition: A CFG is ambiguous if there is at least one string with two or more distinct parse trees
- Ambiguity is not problematic when dealing with flat strings. It is when dealing with language semantics



Detecting and solving Ambiguity

- There is no mechanical way to determine if a grammar is (un)ambiguous (this is an undecidable problem)
- In most practical cases however, it is easy to detect and prove ambiguity.
  E.g., any grammar containing $N \to N\alpha N$ is ambiguous (two parse trees for $N\alpha N\alpha N$).
- How to deal with ambiguities?
  - Modify the grammar to make it unambiguous
  - Handle these ambiguities in the parsing algorithm
- Two common sources of ambiguity in programming languages
  - Expression syntax (operator precedences)
  - Dangling else

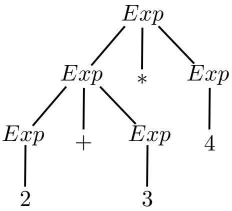
Operator precedence

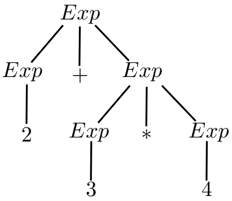
- This expression grammar is ambiguous

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathbf{+}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathbf{-}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathbf{*}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathbf{/}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathbf{num} \\
\mathit{Exp} & \rightarrow & \mathbf{(}\ \mathit{Exp}\ \mathbf{)}
\end{array}
$$

(it contains $N \rightarrow N\alpha N$)

- Parsing of $2 + 3 * 4$:





Operator associativity

- Types of operator associativity:
  - An operator $\oplus$ is left-associative if $a \oplus b \oplus c$ must be evaluated from left to right, i.e., as $(a \oplus b) \oplus c$
  - An operator $\oplus$ is right-associative if $a \oplus b \oplus c$ must be evaluated from right to left, i.e., as $a \oplus (b \oplus c)$
  - An operator $\oplus$ is non-associative if expressions of the form $a \oplus b \oplus c$ are not allowed
- Examples:
  - $-$ and $/$ are typically left-associative
    ($a - b - c = (a - b) - c$)
  - $+$ and $*$ are mathematically associative (left or right).
    By convention, we take them left-associative as well
  - List construction in functional languages is right-associative
  - The assignment operator in C is right-associative
    ($a = b = c$ is equivalent to $a = (b = c)$)
  - In Java, comparison operators are non-associative
    (you can not write $2 < 3 < 4$)

Rewriting ambiguous expression grammars

- Let's consider the following ambiguous grammar:

$$\begin{array}{lcl} E & \rightarrow & E \oplus E \\ E & \rightarrow & \text{num} \end{array}$$

- If $\oplus$ is left-associative, we rewrite it as a left-recursive (a recursive reference only to the left).
- If $\oplus$ is right-associative, we rewrite it as a right-recursive (a recursive reference only to the right).

$\oplus$ left-associative

$$\begin{array}{lcl} E & \rightarrow & E \oplus E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \text{num} \end{array}$$

$\oplus$ right-associative

$$\begin{array}{lcl} E & \rightarrow & E' \oplus E \\ E & \rightarrow & E' \\ E' & \rightarrow & \text{num} \end{array}$$

Mixing operators of different precedence levels

- Introduce a different nonterminal for each precedence level

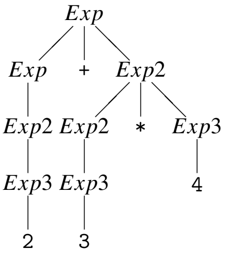
Ambiguous

Exp	→	*Exp* **+** *Exp*
Exp	→	*Exp* **–** *Exp*
Exp	→	*Exp* ***** *Exp*
Exp	→	*Exp* **/** *Exp*
Exp	→	**num**
Exp	→	**(** *Exp* **)**

Non-ambiguous

Exp	→	*Exp* **+** *Term*
Exp	→	*Exp* **–** *Term*
Exp	→	*Term*
Term	→	*Term* ***** *Factor*
Term	→	*Term* **/** *Factor*
Term	→	*Factor*
Factor	→	**num**
Factor	→	**(** *Exp* **)**

Parse tree for $2 + 3 * 4$



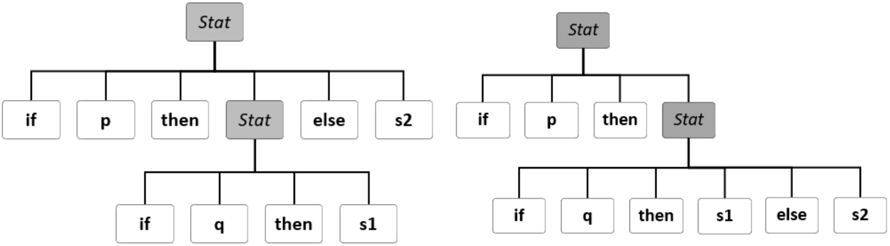
- In this example, the nonterminals *Exp*, *Term* and *Factor* define the application order of the operators in an expression without parentheses.

Dangling else

- Else part of a condition is typically optional

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ else } \mathit{Stat} \\ \mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \end{array}$$

- How to match `if p then if q then s1 else s2`?

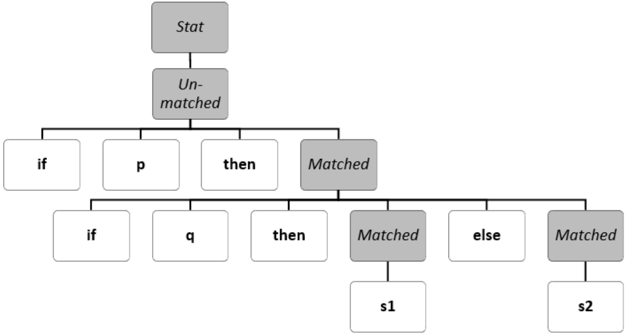


- Convention: `else` matches the closest not previously matched `if`.

Dangling else

- Unambiguous grammar:

$$
\begin{array}{rcl}
\mathit{Stat} & \rightarrow & \mathit{Matched} \mid \mathit{Unmatched} \\
\mathit{Matched} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Matched} \textbf{ else } \mathit{Matched} \\
\mathit{Matched} & \rightarrow & \text{"Any other statement"} \\
\mathit{Unmatched} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \\
\mathit{Unmatched} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Matched} \textbf{ else } \mathit{Unmatched}
\end{array}
$$



End-of-file marker

- Parsers must read not only terminal symbols such as **+** , **−** , **num** but also the end-of-file
- We typically use **$** to represent end of file
- If S is the start symbol of the grammar, then a new start symbol S' is added with the rule $S' \rightarrow S$ **$** .

$$
\begin{array}{rcl}
S & \rightarrow & Exp\ \mathbf{\$} \\
Exp & \rightarrow & Exp\ \mathbf{+}\ Term \\
Exp & \rightarrow & Exp\ \mathbf{-}\ Term \\
Exp & \rightarrow & Term \\
Term & \rightarrow & Term\ \mathbf{*}\ Factor \\
Term & \rightarrow & Term\ \mathbf{/}\ Factor \\
Term & \rightarrow & Factor \\
Factor & \rightarrow & \mathbf{num} \\
Factor & \rightarrow & \mathbf{(}\ Exp\ \mathbf{)}
\end{array}
$$

Non context-free languages

- Some syntactic constructs from typical programming languages cannot be specified with CFG
- Example 1: in C and Java, variables must be declared before their use
  - $L_1 = \{wcw \mid w \text{ is in } (a \mid b)^*\}$ is not context-free
  - As a consequence, one can prove that there is no CFG that checks whether a variable is declared before its use
- Example 2: checking that a function is called with the right number of arguments
  - $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context-free
  - As a consequence, one can prove that there is no CFG that checks whether a function call is consistent with its definition
  - In C, the grammar does not count the number of function arguments

$$\begin{array}{rcl} Stmt & \rightarrow & \textbf{id}\ (Expr\_list) \\ Expr\_list & \rightarrow & Expr\_list, Expr \\ & | & Expr \end{array}$$

- These constructs are typically dealt with during semantic analysis

Backus-Naur Form

- BNF is a text format for describing context-free languages
- We ask you to provide the source grammar for your project in this format
- Example:

```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <factor> "*" <term>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
<digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

- More information:
  http://en.wikipedia.org/wiki/Backus-Naur_form

Syntax analysis

- Goals:
  - Checking that a program is accepted by the context-free grammar
  - Building the parse tree
  - Reporting syntax errors
- Two ways:
  - Top-down: from the start symbol to the word
  - Bottom-up: from the word to the start symbol

Top-down and bottom-up: example

Grammar:

$$\begin{array}{rcl} S & \rightarrow & AB \\ A & \rightarrow & aA \mid \varepsilon \\ B & \rightarrow & b \mid bB \end{array}$$

Top-down parsing of *aaab*

S	
AB	$S \rightarrow AB$
aAB	$A \rightarrow aA$
$aaAB$	$A \rightarrow aA$
$aaaAB$	$A \rightarrow aA$
$aaa\varepsilon B$	$A \rightarrow \varepsilon$
$aaab$	$B \rightarrow b$

Bottom-up parsing of *aaab*

$aaab$	
$aaa\varepsilon b$	(insert ε)
$aaaAb$	$A \rightarrow \varepsilon$
$aaAb$	$A \rightarrow aA$
aAb	$A \rightarrow aA$
Ab	$A \rightarrow aA$
AB	$B \rightarrow b$
S	$S \rightarrow AB$

3.3
Top-down parsing

A naive top-down parser

- A very naive parsing algorithm:
  - Generate all possible parse trees until you get one that matches your input
  - To generate all parse trees:
    1. Start with the root of the parse tree (the start symbol of the grammar)
    2. Choose a non-terminal A at one leaf of the current parse tree
    3. Choose a production having that non-terminal as LHS, eg., $A \rightarrow X_1 X_2 \dots X_k$
    4. Expand the tree by making $X_1, X_2, \dots, X_k$, the children of A.
    5. Repeat at step 2 until all leaves are terminals
    6. Repeat the whole procedure by changing the productions chosen at step 3

  (Note: the choice of the non-terminal in Step 2 is irrevelant for a context-free grammar)
- This algorithm is very inefficient, does not always terminate, etc.

Top-down parsing with backtracking

- Modifications of the previous algorithm:
  1. Depth-first development of the parse tree (corresponding to a left-most derivation)
  2. Process the terminals in the RHS during the development of the tree, checking that they match the input
  3. If they don't at some step, stop expansion and restart at the previous non-terminal with another production rules (backtracking)
- Depth-first can be implemented by storing the unprocessed symbols on a stack
- Because of the left-most derivation, the inputs can be processed from left to right

Backtracking example

$$
\begin{aligned}
S &\rightarrow bab \\
S &\rightarrow bA \\
A &\rightarrow d \\
A &\rightarrow cA
\end{aligned}
$$

$$w = bcd$$

Stack	Inputs	Action
S	bcd	Try $S \rightarrow bab$
bab	bcd	match b
ab	cd	dead-end, backtrack
S	bcd	Try $S \rightarrow bA$
bA	bcd	match b
A	cd	Try $A \rightarrow d$
d	cd	dead-end, backtrack
A	cd	Try $A \rightarrow cA$
cA	cd	match c
A	d	Try $A \rightarrow d$
d	d	match d
		Success!

Top-down parsing with backtracking

- General algorithm (to match a word w):

```
Create a stack with the start symbol
X = POP()
a = GETNEXTTOKEN()
while (True)
    if (X is a nonterminal)
        Pick next rule to expand X → Y1 Y2 ... Yk
        Push Yk, Yk−1, ..., Y1 on the stack
        X = POP()
    elseif (X == $ and a == $)
        Accept the input
    elseif (X == a)
        a = GETNEXTTOKEN()
        X = POP()
    else
        Backtrack
```

- Ok for small grammars but still untractable and very slow for large grammars
- Worst-case exponential time in case of syntax error

Top-down parsing with backtracking

- A top-down parser (even with backtracking) can go into an infinite loop if the grammar is left-recursive:

$$\begin{array}{rcl} Exp & \rightarrow & Exp + Term \mid Term \\ Term & \rightarrow & Term * Factor \mid Factor \\ Factor & \rightarrow & num \mid id \mid (Exp) \end{array}$$

$$w = (id + id) * num$$

Stack	Inputs	Action
Exp	$(id + id) * num$	Try $Exp \rightarrow Exp + Term$
$Exp + Term$	$(id + id) * num$	Try $Exp \rightarrow Exp + Term$
$Exp + Term + Term$	$(id + id) * num$	Try $Exp \rightarrow Exp + Term$
...	...	...

- Such grammars have to be reformulated in order to avoid left-recursion (see later)

Another example

$$
\begin{aligned}
S &\to aSbT \\
S &\to cT \\
S &\to d \\
T &\to aT \\
T &\to bS \\
T &\to c
\end{aligned}
$$

$w = accbbadbc$

Stack	Inputs	Action
S	$accbbadbc$	Try $S \to aSbT$
$aSbT$	$accbbadbc$	match a
SbT	$accbbadbc$	Try $S \to aSbT$
$aSbTbT$	$accbbadbc$	match a
$SbTbT$	$ccbbadbc$	Try $S \to cT$
$cTbTbT$	$ccbbadbc$	match c
$TbTbT$	$cbbadbc$	Try $T \to c$
$cbTbT$	$cbbadbc$	match cb
TbT	$badbc$	Try $T \to bS$
$bSbT$	$badbc$	match b
SbT	$adbc$	Try $S \to aSbT$
$aSbT$	$adbc$	match a
...	...	...
c	c	match c
		Success!

Predictive parsing

- Predictive parser:
  - In the previous example, the production rule to apply can be predicted based solely on the next input symbol and the current nonterminal
  - Much faster than backtracking but this trick works only for some specific grammars
- Grammars for which top-down predictive parsing is possible by looking at the next symbol are called $\boxed{LL(1) \text{ grammars}}$:
  - L: left-to-right scan of the tokens
  - L: leftmost derivation
  - (1): one token of lookahead
- Predicted rules are stored in a parsing table M:
  - $M[X, a]$ stores the rule to apply when the nonterminal X is on the stack and the next input terminal is a

Example: parse table

$$S \rightarrow E\$$$
$$E \rightarrow \texttt{int}$$
$$E \rightarrow (E\ Op\ E)$$
$$Op \rightarrow +$$
$$Op \rightarrow *$$

	int	(	)	+	*	$
S	E$	E$				
E	int	(E Op E)				
Op				+	*	

(Keith Schwarz)

Example: successfull parsing

1. S → E`$`
2. E → `int`
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

S	`(int + (int * int))$`
E`$`	`(int + (int * int))$`
`(`E Op E`)$`	`(int + (int * int))$`
E Op E`)$`	`int + (int * int))$`
`int` Op E`)$`	`int + (int * int))$`
Op E`)$`	`+ (int * int))$`
`+` E`)$`	`+ (int * int))$`
E`)$`	`(int * int))$`
`(`E Op E`))$`	`(int * int))$`
E Op E`))$`	`int * int))$`
`int` Op E`))$`	`int * int))$`
Op E`))$`	`* int))$`
`*` E`))$`	`* int))$`
E`))$`	`int))$`
`int))$`	`int))$`
`))$`	`))$`
`)$`	`)$`
`$`	`$`

(Keith Schwarz)

Example: erroneous parsing

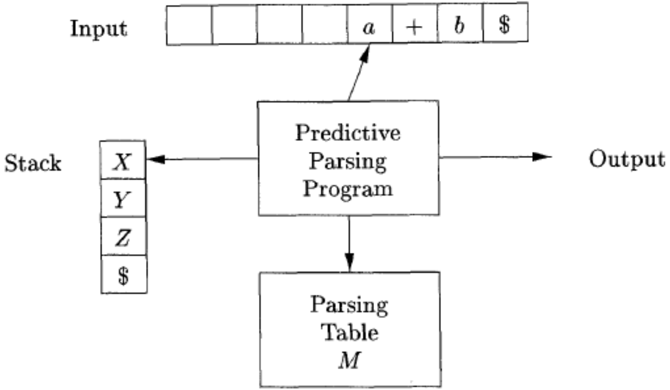
1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	`(int (int))$`
E$	`(int (int))$`
(E Op E)$	`(int (int))$`
E Op E)$	`int (int))$`
int Op E)$	`int (int))$`
Op E)$	`(int))$`

(Keith Schwarz)

Table-driven predictive parser



(Dragonbook)

Table-driven predictive parser

```
Create a stack with the start symbol
X = POP()
a = GETNEXTTOKEN()
while (True)
    if (X is a nonterminal)
        if (M[X, a] == NULL)
            Error
        elseif (M[X, a] == X → Y1 Y2 ... Yk)
            Push Yk, Yk−1, ..., Y1 on the stack
            X = POP()
    elseif (X == $ and a == $)
        Accept the input
    elseif (X == a)
        a = GETNEXTTOKEN()
        X = POP()
    else
        Error
```

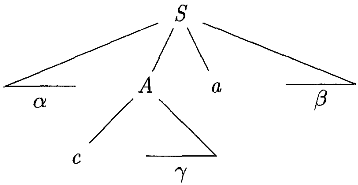
$LL(1)$ grammars and parsing

Three questions we need to address:

- How to build the table for a given grammar?
- How to know if a grammar is $LL(1)$?
- How to change a grammar to make it $LL(1)$?

Building the table

- It is useful to define three functions
  (with A a nonterminal and α any sequence of grammar symbols):
  - *Nullable*(α) is true if $\alpha \stackrel{*}{\Rightarrow} \varepsilon$
  - *First*(α) returns the set of terminals c such that $\alpha \stackrel{*}{\Rightarrow} c\gamma$ for some (possibly empty) sequence γ of grammar symbols
  - *Follow*(A) returns the set of terminals a such that $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, where α and β are (possibly empty) sequences of grammar symbols



($c \in First(A)$ and $a \in Follow(A)$)

Example

$$
\begin{array}{rcl}
C & \rightarrow & P\ F\ \textbf{class}\ \ \textbf{id}\ X\ Y\ \$ \\
P & \rightarrow & \textbf{public}\ \mid \varepsilon \\
F & \rightarrow & \textbf{final}\ \mid \varepsilon \\
X & \rightarrow & \textbf{extends}\ \ \textbf{id}\ \mid \varepsilon \\
Y & \rightarrow & \textbf{implements}\ I \mid \varepsilon \\
I & \rightarrow & \textbf{id}\ J \\
J & \rightarrow & \textbf{,}\ I \mid \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
First(C) & = & \{\ \textbf{public}\ ,\ \textbf{final}\ ,\ \textbf{class}\ \} \\
First(P) & = & \{\ \textbf{public}\ \} \\
First(F) & = & \{\ \textbf{final}\ \} \\
First(X) & = & \{\ \textbf{extends}\ \} \\
First(I) & = & \{\ \textbf{id}\ \} \\
First(J) & = & \{'\ \textbf{,}\ '\}
\end{array}
$$

$$
\begin{array}{rcl}
Follow(C) & = & \{\} \\
Follow(P) & = & \{\ \textbf{final}\ ,\ \textbf{class}\ \} \\
Follow(F) & = & \{\ \textbf{class}\ \} \\
Follow(X) & = & \{\ \textbf{implements}\ , \$\} \\
Follow(Y) & = & \{\$\} \\
Follow(I) & = & \{\$\} \\
Follow(J) & = & \{\$\}
\end{array}
$$

(Carnahan)

Building the table from *First*, *Follow*, and *Nullable*

To construct the table:

1. Start with the empty table
2. For each production $A \to \alpha$:
   2.1 add $A \to \alpha$ to $M[A, a]$ for each terminal a in $First(\alpha)$
   2.2 if $Nullable(\alpha)$ then add $A \to \alpha$ to $M[A, a]$ for each a in $Follow(A)$

Illustration:

	public	**final**	**class**	$\cdots$
C	P F **class** **id** X Y \$	P F **class** **id** X Y \$	P F **class** **id** X Y \$	
P	**public**	ε	ε	
F		**final**	ε	
$\cdots$				

$LL(1)$ grammars

- Three situations:
  - $M[A, a]$ is empty: no production is appropriate. We can not parse the sentence and have to report a syntax error
  - $M[A, a]$ contains one entry: perfect!
  - $M[A, a]$ contains two entries: the grammar is not appropriate for predictive parsing (with one token lookahead)
- Definition: A grammar is $LL(1)$ if its parsing table contains at most one entry in each cell or, equivalently, if for all production pairs $A \to \alpha \mid \beta$
  - $First(\alpha) \cap First(\beta) = \emptyset$,
  - $Nullable(\alpha)$ and $Nullable(\beta)$ are not both true,
  - if $Nullable(\beta)$, then $First(\alpha) \cap Follow(A) = \emptyset$

Example of a non $LL(1)$ grammar

$$
\begin{array}{rcl}
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathbf{+}\ \mathit{Term} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathbf{-}\ \mathit{Term} \\
\mathit{Exp} & \rightarrow & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term}\ \mathbf{*}\ \mathit{Factor} \\
\mathit{Term} & \rightarrow & \mathit{Term}\ \mathbf{/}\ \mathit{Factor} \\
\mathit{Term} & \rightarrow & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \mathbf{num} \\
\mathit{Factor} & \rightarrow & \mathbf{(}\ \mathit{Exp}\ \mathbf{)}
\end{array}
$$

($\mathit{First}(\alpha)$ is the same for all RHS α of the productions for *Exp* and *Term*)

Computing *Nullable*

Algorithm to compute *Nullable* for all grammar symbols:

Initialize *Nullable* to *False*.
repeat
  for each production $X \rightarrow Y_1 Y_2 \ldots Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
      $Nullable(X) = True$
until *Nullable* did not change in this iteration.

Algorithm to compute *Nullable* for any string $\alpha = X_1 X_2 \ldots X_k$:

if $X_1 \ldots X_k$ are all nullable
  $Nullable(\alpha) = True$
else
  $Nullable(\alpha) = False$

Computing *First*

Algorithm to compute *First* for all grammar symbols:

Initialize *First* to empty sets.
for each terminal z
    $\mathit{First}(z) = \{z\}$
repeat
    for each production $X \rightarrow Y_1 Y_2 \ldots Y_k$
        for $i = 1$ **to** k
            if $Y_1 \ldots Y_{i-1}$ are all nullable (or $i = 1$)
                $\mathit{First}(X) = \mathit{First}(X) \cup \mathit{First}(Y_i)$
until *First* did not change in this iteration.

Algorithm to compute *First* for any string $\alpha = X_1 X_2 \ldots X_k$:

Initialize $\mathit{First}(\alpha) = \emptyset$
for $i = 1$ **to** k
    if $X_1 \ldots X_{i-1}$ are all nullable (or $i = 1$)
        $\mathit{First}(\alpha) = \mathit{First}(\alpha) \cup \mathit{First}(X_i)$

Computing *Follow*

Algorithm to compute *Follow* for all nonterminal symbols:

Initialize *Follow* to empty sets.
repeat
  for each production $X \rightarrow Y_1 Y_2 \ldots Y_k$
    for $i = 1$ **to** k, **for** $j = i + 1$ **to** k
      if $Y_{i+1} \ldots Y_k$ are all nullable (or $i = k$)
        $\mathit{Follow}(Y_i) = \mathit{Follow}(Y_i) \cup \mathit{Follow}(X)$
      if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or $i + 1 = j$)
        $\mathit{Follow}(Y_i) = \mathit{Follow}(Y_i) \cup \mathit{First}(Y_j)$
until *Follow* did not change in this iteration.

Example

Compute the parsing table for the following grammar:

$$
\begin{array}{rcl}
S & \rightarrow & E\$ \\
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \\
E' & \rightarrow & -\ T\ E' \\
E' & \rightarrow & \varepsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \\
T' & \rightarrow & /\ F\ T' \\
T' & \rightarrow & \varepsilon \\
F & \rightarrow & \mathbf{id} \\
F & \rightarrow & \mathbf{num} \\
F & \rightarrow & (\ E\ )
\end{array}
$$

Example

Nonterminals	*Nullable*	*First*	*Follow*
S	False	$\{(,$ **id** $,$ **num** $\}$	$\emptyset$
E	False	$\{(,$ **id** $,$ **num** $\}$	$\{), \$\}$
E'	True	$\{+, -\}$	$\{), \$\}$
T	False	$\{(,$ **id** $,$ **num** $\}$	$\{), +, -, \$\}$
T'	True	$\{*, /\}$	$\{), +, -, \$\}$
F	False	$\{(,$ **id** $,$ **num** $\}$	$\{), *, /, +, -, \$\}$

	$+$	$*$	**id**	(	)	\$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$			$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F			$F \rightarrow$ **id**	$F \rightarrow (E)$		

($-$,/, and **num** are treated similarly)

Transforming a grammar for $LL(1)$ parsing

- Ambiguous grammars are not $LL(1)$ but unambiguous grammars are not necessarily $LL(1)$
- Having a non-$LL(1)$ unambiguous grammar for a language does not mean that this language is not $LL(1)$.
- But there are languages for which there exist unambiguous context-free grammars but no $LL(1)$ grammar.

- We will see two grammar transformations that improve the chance to get a $LL(1)$ grammar:
  - Elimination of left-recursion
  - Left-factorization

Left-recursion

- The following expression grammar is unambiguous but it is not $LL(1)$:

$$
\begin{array}{rcl}
Exp & \rightarrow & Exp \; \mathbf{+} \; Term \\
Exp & \rightarrow & Exp \; \mathbf{-} \; Term \\
Exp & \rightarrow & Term \\
Term & \rightarrow & Term \; \mathbf{*} \; Factor \\
Term & \rightarrow & Term \; \mathbf{/} \; Factor \\
Term & \rightarrow & Factor \\
Factor & \rightarrow & \mathbf{num} \\
Factor & \rightarrow & \mathbf{(} \; Exp \; \mathbf{)}
\end{array}
$$

- Indeed, $First(\alpha)$ is the same for all RHS α of the productions for *Exp* et *Term*
- This is a consequence of *left-recursion*.

Left-recursion

- Recursive productions are productions defined in terms of themselves. Examples: $A \rightarrow Ab$ or $A \rightarrow bA$.
- When the recursive nonterminal is at the left (resp. right), the production is said to be left-recursive (resp. right-recursive).
- Left-recursive productions can be rewritten with right-recursive productions
- Example:

$$\begin{array}{rcl} N & \rightarrow & N\alpha_1 \\ & \vdots & \\ N & \rightarrow & N\alpha_m \\ N & \rightarrow & \beta_1 \\ & \vdots & \\ N & \rightarrow & \beta_n \end{array} \quad \Leftrightarrow \quad \begin{array}{rcl} N & \rightarrow & \beta_1 N' \\ & \vdots & \\ N & \rightarrow & \beta_n N' \\ N' & \rightarrow & \alpha_1 N' \\ & \vdots & \\ N' & \rightarrow & \alpha_m N' \\ N' & \rightarrow & \varepsilon \end{array}$$

Right-recursive expression grammar

$$
\begin{array}{rcl}
Exp & \rightarrow & Exp + Term \\
Exp & \rightarrow & Exp - Term \\
Exp & \rightarrow & Term \\
 & & \\
Term & \rightarrow & Term * Factor \\
Term & \rightarrow & Term \; / \; Factor \\
Term & \rightarrow & Factor \\
 & & \\
Factor & \rightarrow & \mathbf{num} \\
Factor & \rightarrow & (\; Exp \;)
\end{array}
\quad \Leftrightarrow \quad
\begin{array}{rcl}
Exp & \rightarrow & Term \; Exp' \\
Exp' & \rightarrow & + \; Term \; Exp' \\
Exp' & \rightarrow & - \; Term \; Exp' \\
Exp' & \rightarrow & \varepsilon \\
 & & \\
Term & \rightarrow & Factor \; Term' \\
Term' & \rightarrow & * \; Factor \; Term' \\
Term' & \rightarrow & / \; Factor \; Term' \\
Term' & \rightarrow & \varepsilon \\
 & & \\
Factor & \rightarrow & \mathbf{num} \\
Factor & \rightarrow & (\; Exp \;)
\end{array}
$$

Left-factorisation

- The RHS of these two productions have the same *First* set.

$$
\begin{array}{rcl}
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ else } \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat}
\end{array}
$$

- The problem can be solved by left factorising the grammar:

$$
\begin{array}{rcl}
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat}\ \mathit{ElseStat} \\
\mathit{ElseStat} & \rightarrow & \textbf{else } \mathit{Stat} \\
\mathit{ElseStat} & \rightarrow & \varepsilon
\end{array}
$$

- Note
  - The resulting grammar is ambiguous and the parsing table will contain two rules for $M[\mathit{ElseStat}, \textbf{else}]$ (because $\textbf{else} \in \mathit{Follow}(\mathit{ElseStat})$ and $\textbf{else} \in \mathit{First}(\textbf{else}\ \mathit{Stat})$)
  - Ambiguity can be solved in this case by letting $M[\mathit{ElseStat}, \textbf{else}] = \{\mathit{ElseStat} \rightarrow \textbf{else}\ \mathit{Stat}\}$.

Hidden left-factors and hidden left recursion

- Sometimes, left-factors or left recursion are hidden
- Examples:
  - The following grammar:

$$\begin{array}{rcl} A & \rightarrow & da \mid acB \\ B & \rightarrow & abB \mid daA \mid Af \end{array}$$

    has two overlapping productions: $B \rightarrow daA$ and $B \stackrel{*}{\Rightarrow} daf$.
  - The following grammar:

$$\begin{array}{rcl} S & \rightarrow & Tu \mid wx \\ T & \rightarrow & Sq \mid vvS \end{array}$$

    has left recursion on T ($T \stackrel{*}{\Rightarrow} Tuq$)
- Solution: expand the production rules by substitution to make left-recursion or left factors visible and then eliminate them

Summary

Construction of a $LL(1)$ parser from a CFG grammar

- Eliminate ambiguity
- Eliminate left recursion
- Left factorization
- Add an extra start production $S' \rightarrow S\$$ to the grammar
- Calculate *First* for every production and *Follow* for every nonterminal
- Calculate the parsing table
- Check that the grammar is $LL(1)$

Recursive implementation

- From the parsing table, it is easy to implement a predictive parser recursively (with one function per nonterminal)

$$
\begin{array}{rcl}
T' & \rightarrow & T\$ \\
T & \rightarrow & R \\
T & \rightarrow & aTc \\
R & \rightarrow & \epsilon \\
R & \rightarrow & bR
\end{array}
$$

	a	b	c	$\$$
T'	$T' \rightarrow T\$$	$T' \rightarrow T\$$		$T' \rightarrow T\$$
T	$T \rightarrow aTc$	$T \rightarrow R$	$T \rightarrow R$	$T \rightarrow R$
R		$R \rightarrow bR$	$R \rightarrow \epsilon$	$R \rightarrow \epsilon$

```
function parseT'() =
  if next = 'a' or next = 'b' or next = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if next = 'b' or next = 'c' or next = '$' then
    parseR()
  else if next = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if next = 'c' or next = '$' then
    (* do nothing *)
  else if next = 'b' then
    match('b') ; parseR()
  else reportError()
```

(Mogensen)