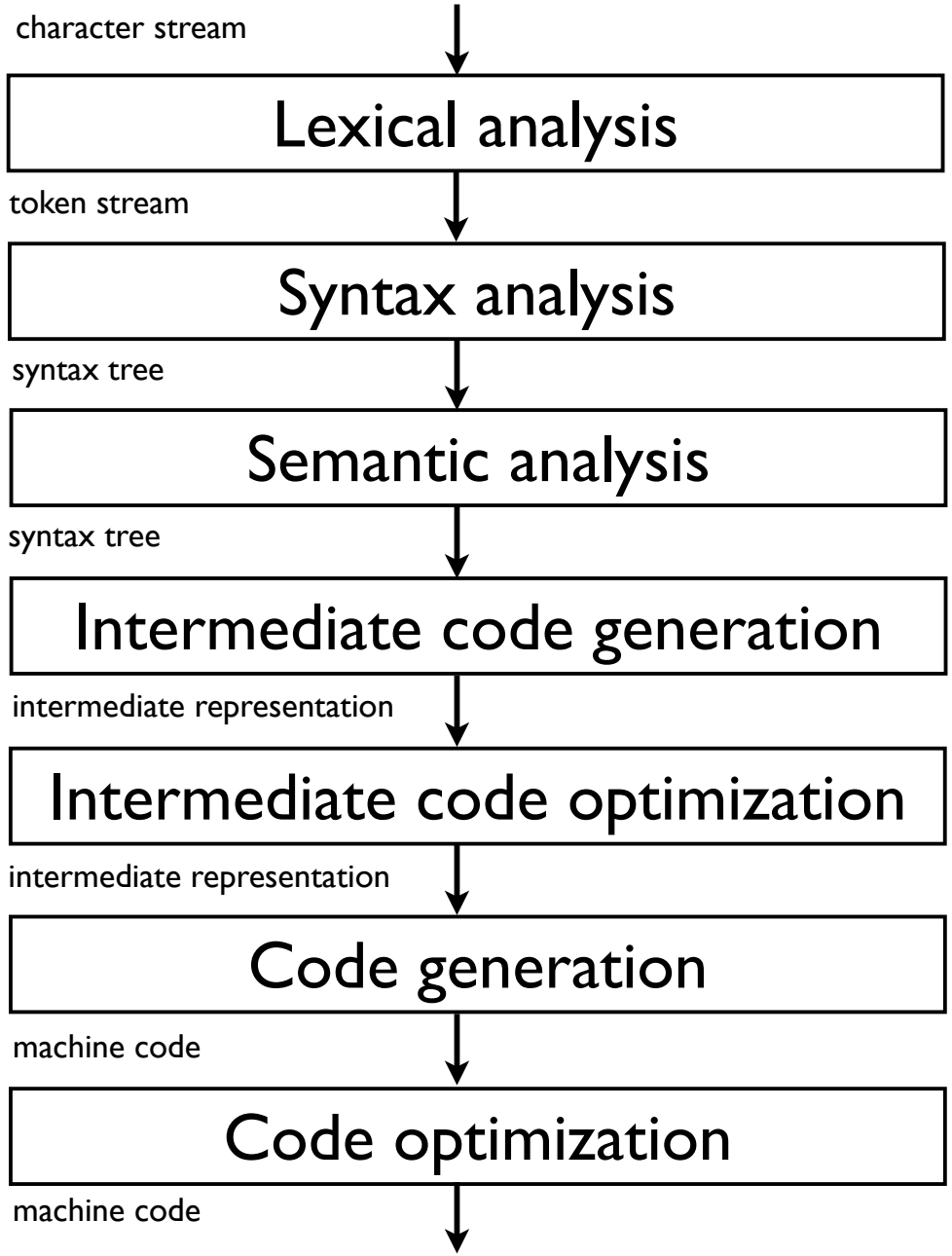
Part 3

Syntax analysis

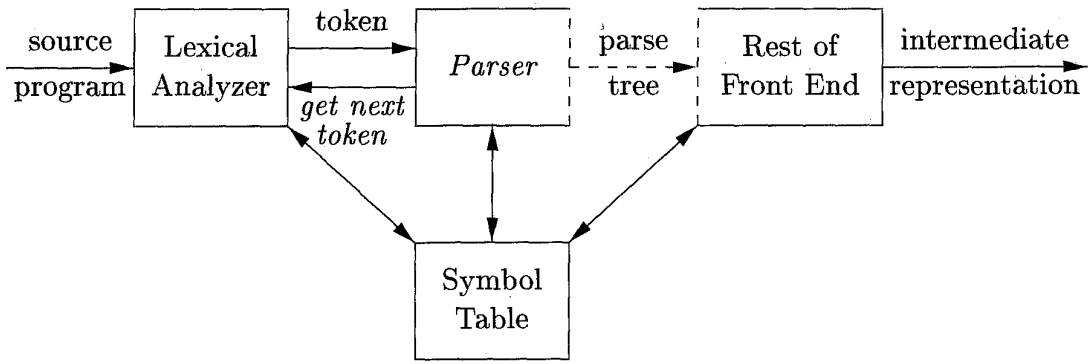
Outline

1. Introduction
2. Context-free grammar
3. Top-down parsing
4. Bottom-up parsing
5. Conclusion and some practical considerations

Structure of a compiler

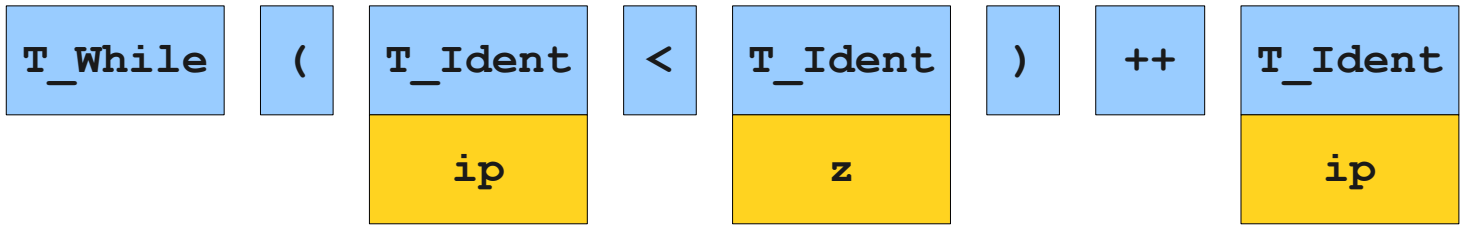


Syntax analysis



- Goals:
  - 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

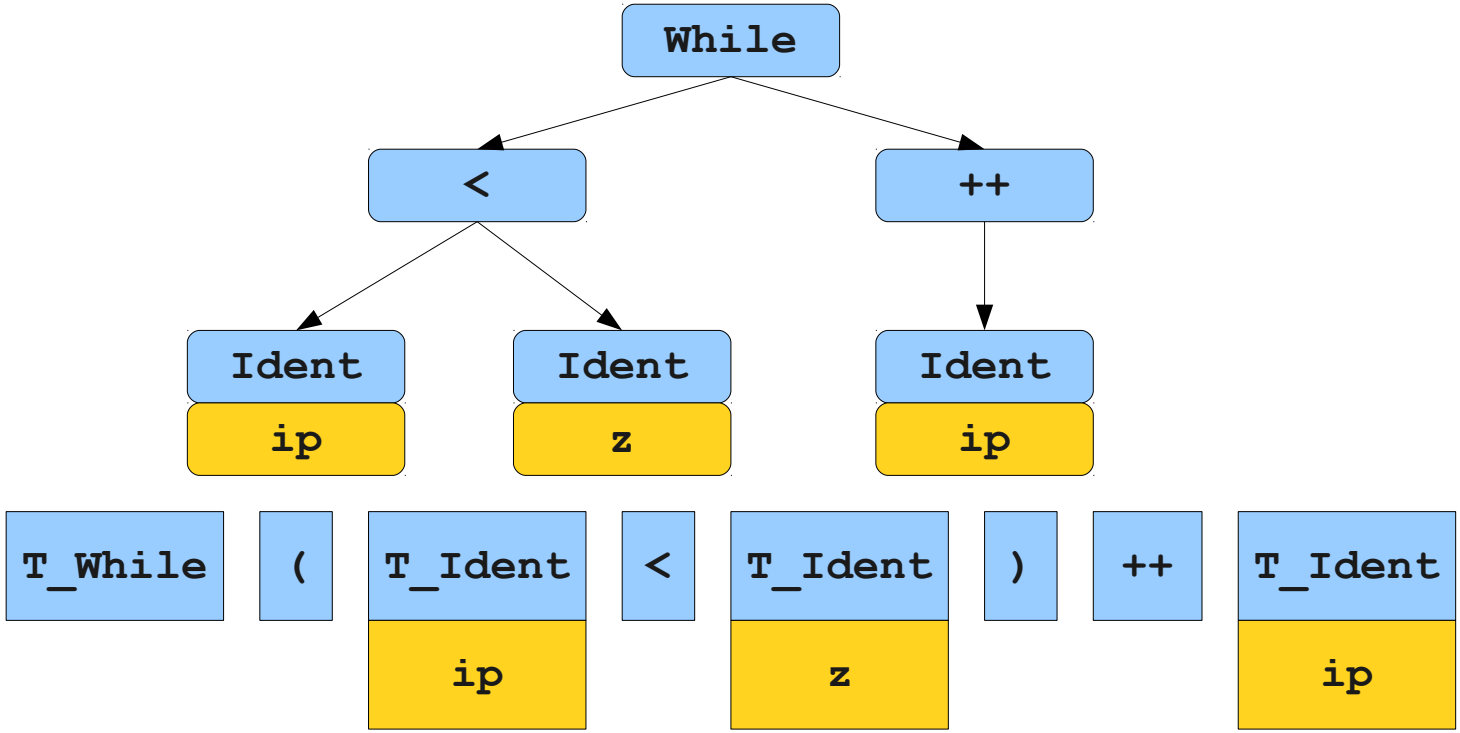


w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

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

(Keith Schwarz)

Example



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

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

(Keith Schwarz)

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: $A \to \alpha|\beta$
- Example: $\Sigma = \{a, b, c\}$, $V - \Sigma = \{S, R\}$, $R =$

$$\begin{aligned} S &\to R \\ S &\to aSc \\ R &\to \epsilon \\ R &\to RbR \end{aligned}$$

Reminder: derivation and language

Definitions:

- v can be *derived in one step* from u by G (noted $v \Rightarrow u$) iff $u = xu'y$, $v = xv'y$, and $u' \rightarrow v'$
- v can be *derived in several steps* from u by G (noted $v \overset{*}{\Rightarrow} u$) iff $\exists k \geq 0$ and $v_0 \dots v_k \in V^+$ 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:

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

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 $uXw \rightarrow uvw$, where u, v, w are arbitrary strings of symbols in V, with v non-null, and X a single nonterminal
- Type 2: context-free grammars (CFG)
  - productions of the form $X \rightarrow v$ where v 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 $X \rightarrow \epsilon$ 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:
  - $L = \{a^n b^n | n \geq 0\}$
  - Balanced parentheses
    $L = \{\epsilon, (), (()), ()(), ((())), (())() \ldots\}$
  - Scheme programs
    $L = \{1, 2, 3, \ldots, (lambda(x)(+x1))\}$
- 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

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

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \textbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Stat}; \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ Else } \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat}
\end{array}
$$

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
- 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

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

$w = accacbb$

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

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

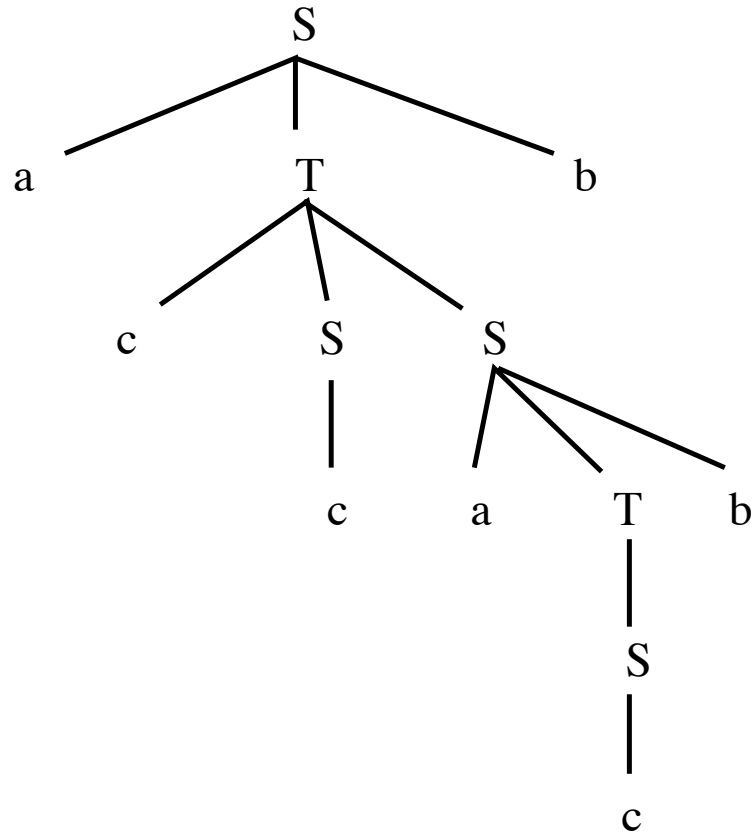
Parse tree

- A parse tree abstracts the order of application of the rules
  - Each interior node represents the application of a production
  - For a rule $A \to X_1 X_2 \dots X_k$, the interior node is labeled by A and the children from left to right by $X_1, X_2, \dots, X_k$.
  - 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

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

$$w = accacbb$$

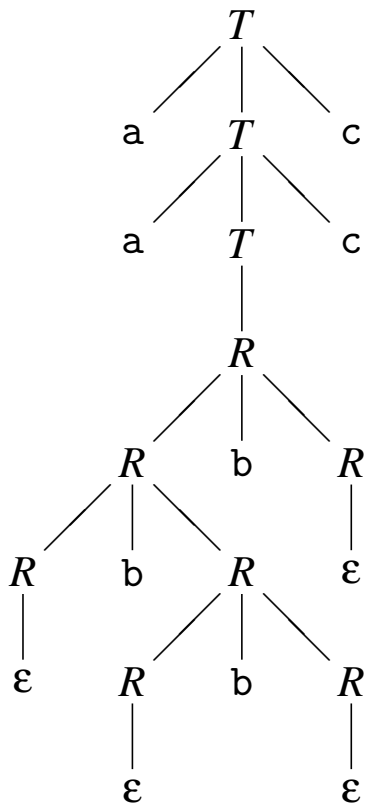


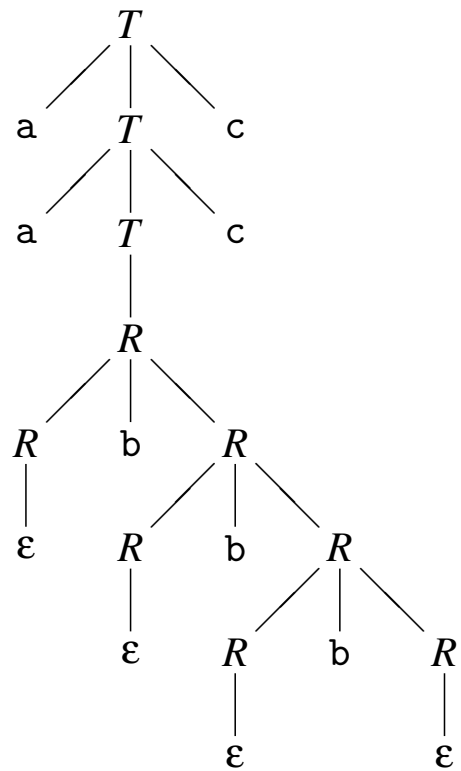
Left-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow accSb \Rightarrow$
$accaTbb \Rightarrow accaSbb \Rightarrow accacbb$

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

Parse tree

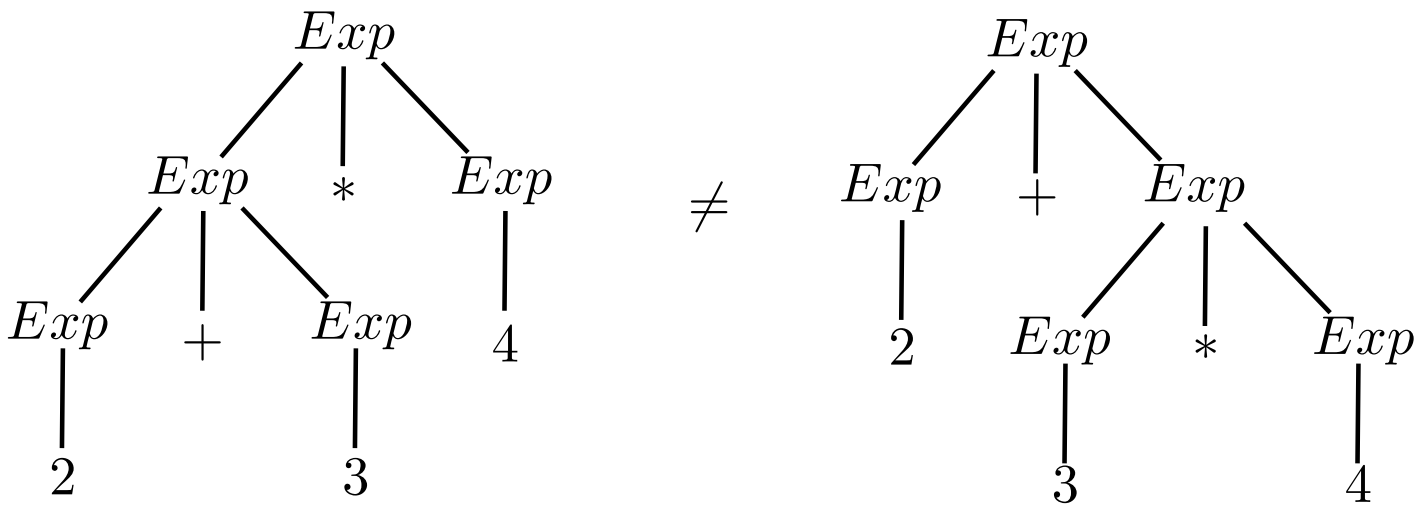
$$
\begin{aligned}
T &\rightarrow R \\
T &\rightarrow aTc \\
R &\rightarrow \epsilon \\
R &\rightarrow RbR
\end{aligned}
$$





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 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

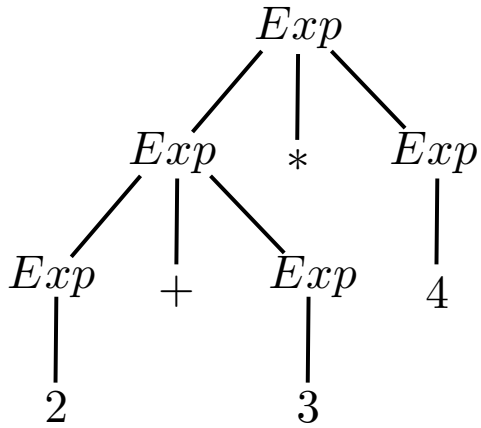
Operator precedence

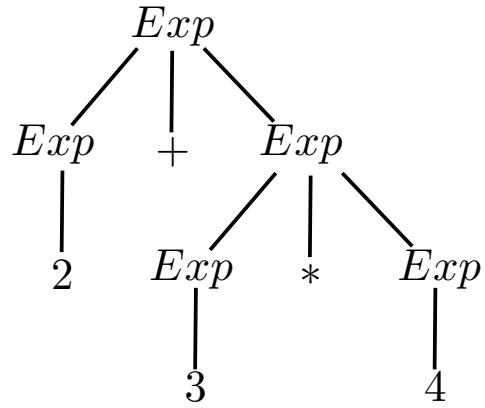
- This expression grammar is ambiguous

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

(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
  - $+$ and $*$ are mathematically associative (left or right). By convention, we take them left-associative as well
  - List construction in functional languages is right-associative
  - Arrows operator in C is right-associative (`a->b->c` is equivalent to `a->(b->c)`)
  - In Pascal, comparison operators are non-associative (you can not write $2 < 3 < 4$)

Rewriting ambiguous expression grammars

- Let's consider the following ambiguous grammar:

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

- 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

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

$\oplus$ right-associative

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

Mixing operators of different precedence levels

- Introduce a different nonterminal for each precedence level

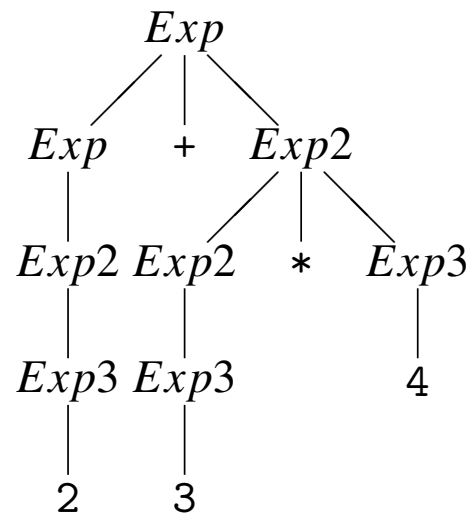
Ambiguous

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

Non-ambiguous

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \text{num} \\
Exp3 & \rightarrow & (Exp)
\end{array}
$$

Parse tree for $2 + 3 * 4$



Dangling else

- Else part of a condition is typically optional

$$\begin{aligned} \mathit{Stat} &\rightarrow \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}\ \textbf{Else}\ \mathit{Stat} \\ \mathit{Stat} &\rightarrow \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat} \end{aligned}$$

- How to match `if p then if q then s1 else s2`?
- Convention: `else` matches the closest not previously matched `if`.
- Unambiguous grammar:

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

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 following rules $S' \to S\$$.

$$
\begin{array}{rcl}
S & \to & Exp\$ \\
Exp & \to & Exp + Exp2 \\
Exp & \to & Exp - Exp2 \\
Exp & \to & Exp2 \\
Exp2 & \to & Exp2 * Exp3 \\
Exp2 & \to & Exp2 / Exp3 \\
Exp2 & \to & Exp3 \\
Exp3 & \to & \text{num} \\
Exp3 & \to & (Exp)
\end{array}
$$

Non-context free languages

- Some syntactic constructs from typical programming languages cannot be specified with CFG
- Example 1: ensuring that a variable is declared before its use
  - $L_1 = \{wcw | w \text{ is in } (a|b)^*\}$ is not context-free
  - In C and Java, there is one token for all identifiers
- Example 2: checking that a function is called with the right number of arguments
  - $L_2 = \{a^n b^m c^n d^m | n \geq 1 \text{ and } m \geq 1\}$ is not context-free
  - In C, the grammar does not count the number of function arguments

$$\begin{aligned} \textit{stmt} &\rightarrow \textbf{id}\ (\textit{expr\_list}) \\ \textit{expr\_list} &\rightarrow \textit{expr\_list}, \textit{expr} \\ &\mid \textit{expr} \end{aligned}$$

- These constructs are typically dealt with during semantic analysis

Backus-Naur Form

- 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`

Outline

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:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA|\epsilon \\
B &\rightarrow b|bB
\end{aligned}
$$

Top-down parsing of *aaab*

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

Bottom-up parsing of *aaab*

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

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., $A \rightarrow X_1 X_2 \ldots X_k$
    4. Expand the tree by making X_1,X_2,. . . ,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

Another example

$$
\begin{array}{lcl}
S & \rightarrow & aSbT \\
S & \rightarrow & cT \\
S & \rightarrow & d \\
T & \rightarrow & aT \\
T & \rightarrow & bS \\
T & \rightarrow & c
\end{array}
$$

$w = accbbadbc$

Stack	Inputs	Action
S	$accbbadbc$	Try $S \rightarrow aSbT$
$aSbT$	$accbbadbc$	match a
SbT	$accbbadbc$	Try $S \rightarrow aSbT$
$aSbTbT$	$accbbadbc$	match a
$SbTbT$	$ccbbadbc$	Try $S \rightarrow cT$
$cTbTbT$	$ccbbadbc$	match c
$TbTbT$	$cbbadbc$	Try $T \rightarrow c$
$cbTbT$	$cbbadbc$	match cb
TbT	$badbc$	Try $T \rightarrow bS$
$bSbT$	$badbc$	match b
SbT	$adbc$	Try $S \rightarrow aSbT$
$aSbT$	$adbc$	match a
$\ldots$	$\ldots$	$\ldots$
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 $LL(1)$ 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 → E$
E → int
E → (E Op E)
Op → +
Op → *

	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

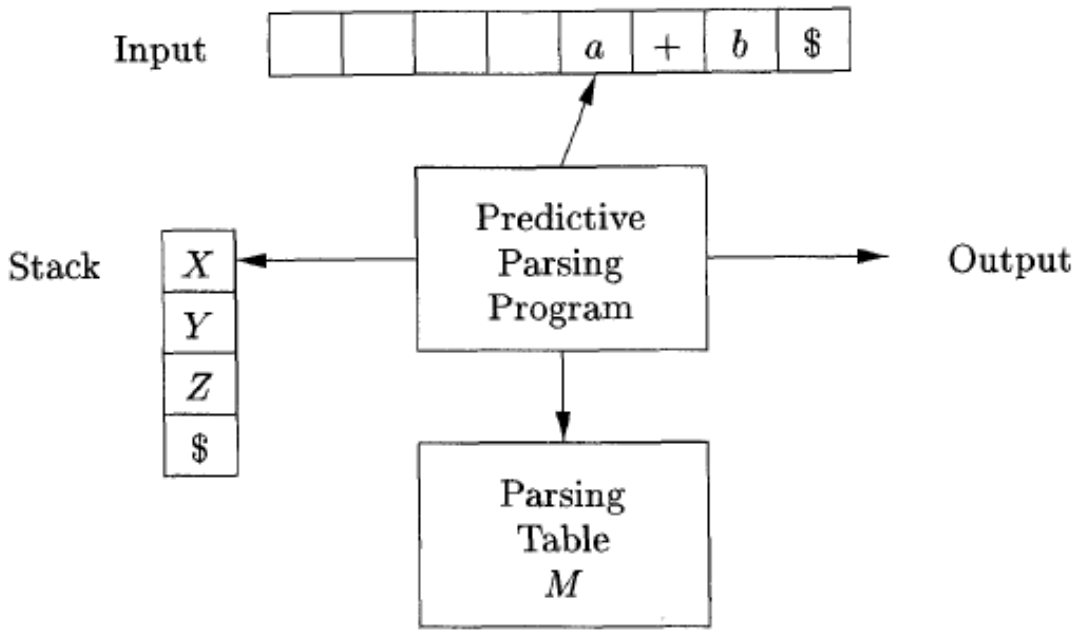
1. S → E$
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	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 ($M[X, a] == X \rightarrow Y_1 Y_2 \ldots Y_k$)
            Push $Y_k, Y_{k-1}, \ldots, Y_1$ on the stack
            X = POP()
    elseif ($X ==$ \$ and $a ==$ \$)
        Accept the input
    elseif ($X == a$)
        a = GETNEXTTOKEN()
        X = POP()
    else
        Error

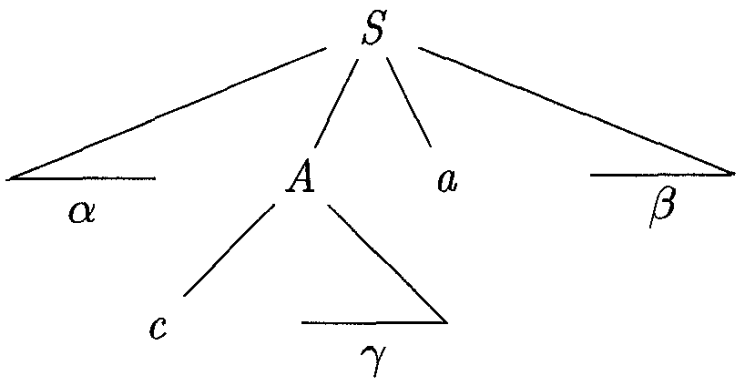
$LL(1)$ grammars and parsing

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 $\alpha \stackrel{*}{\Rightarrow} \epsilon$
  - *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))

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

To construct the table:

- Start with the empty table
- For each production $A \to \alpha$:
  - add $A \to \alpha$ to $M[A, a]$ for each terminal a in $First(\alpha)$
  - If $Nullable(\alpha)$, add $A \to \alpha$ to $M[A, a]$ for each a in $Follow(A)$

First rule is obvious. Illustration of the second rule:

$$\begin{array}{lcl} S & \to & Ab \\ A & \to & c \\ A & \to & \epsilon \end{array} \qquad \begin{array}{rcl} Nullable(A) & = & True \\ First(A) & = & \{c\} \\ Follow(A) & = & \{b\} \end{array} \qquad M[A, b] = A \to \epsilon$$

$LL(1)$ 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 $A \rightarrow \alpha|\beta$
  - $\mathit{First}(\alpha) \cap \mathit{First}(\beta) = \emptyset$,
  - $\mathit{Nullable}(\alpha)$ and $\mathit{Nullable}(\beta)$ are not both true,
  - if $\mathit{Nullable}(\beta)$, then $\mathit{First}(\alpha) \cap \mathit{Follow}(A) = \emptyset$
- Example of a non $LL(1)$ grammar:

$$\begin{aligned} S &\rightarrow Ab \\ A &\rightarrow b \\ A &\rightarrow \epsilon \end{aligned}$$

Computing *Nullable*

Algorithm to compute *Nullable* for all grammar symbols

Initialize *Nullable* to *False*.
repeat
    for each production $X \to Y_1 Y_2 \ldots Y_k$
        if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
            $\mathit{Nullable}(X) = \mathit{True}$
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)
    $\mathit{Nullable}(\alpha) = \mathit{True}$
else
    $\mathit{Nullable}(\alpha) = \mathit{False}$

Computing *First*

Algorithm to compute *First* for all grammar symbols

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

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

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

Computing *Follow*

To compute *Follow* for all nonterminal symbols

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

Example

Compute the parsing table for the following grammar:

$$
\begin{array}{rcl}
S & \rightarrow & E\$ \\
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \\
E' & \rightarrow & -TE' \\
E' & \rightarrow & \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \\
T' & \rightarrow & /FT' \\
T' & \rightarrow & \epsilon \\
F & \rightarrow & \textbf{id} \\
F & \rightarrow & \textbf{num} \\
F & \rightarrow & (E)
\end{array}
$$

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			$S \to E\$$	$S \to E\$$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \epsilon$	$E' \to \epsilon$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to \epsilon$	$T' \to *FT'$			$T' \to \epsilon$	$T' \to \epsilon$
F			$F \to$ **id**	$F \to (E)$		

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

$LL(1)$ parsing summary so far

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

- Eliminate ambiguity
- 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 $LL(1)$

Next course:

- Transformations of a grammar to make it $LL(1)$
- Recursive implementation of the predictive parser
- Bottom-up parsing techniques

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 $LL(1)$:

$$
\begin{array}{rcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (Exp)
\end{array}
$$

- Indeed, $First(\alpha)$ is the same for all RHS α of the productions for Exp et $Exp2$
- This is a consequence of *left-recursion*.

Left-recursion

- Recursive productions are productions defined in terms of themselves. Examples: $A \rightarrow Ab$ ou $A \rightarrow bA$.
- 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:

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

Right-recursive expression grammar

$$
\begin{array}{rcl}
Exp & \to & Exp + Exp2 \\
Exp & \to & Exp - Exp2 \\
Exp & \to & Exp2 \\
Exp2 & \to & Exp2 * Exp3 \\
Exp2 & \to & Exp2 / Exp3 \\
Exp2 & \to & Exp3 \\
Exp3 & \to & \textbf{num} \\
Exp3 & \to & (Exp)
\end{array}
\quad \Leftrightarrow \quad
\begin{array}{rcl}
Exp & \to & Exp2\,Exp' \\
Exp' & \to & +\,Exp2\,Exp' \\
Exp' & \to & -\,Exp2\,Exp' \\
Exp' & \to & \epsilon \\
Exp2 & \to & Exp3\,Exp2' \\
Exp2' & \to & *\,Exp3\,Exp2' \\
Exp2' & \to & /\,Exp3\,Exp2' \\
Exp2' & \to & \epsilon \\
Exp3 & \to & \textbf{num} \\
Exp3 & \to & (Exp)
\end{array}
$$

Left-factorisation

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

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

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

$$\begin{aligned} \mathit{Stat} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat}\ \mathit{ElseStat} \\ \mathit{ElseStat} &\rightarrow \textbf{else } \mathit{Stat} \\ \mathit{ElseStat} &\rightarrow \epsilon \end{aligned}$$

- 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:

$$\begin{aligned} A &\rightarrow da|acB \\ B &\rightarrow abB|daA|Af \end{aligned}$$

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

$$\begin{aligned} S &\rightarrow Tu|wx \\ T &\rightarrow Sq|vvS \end{aligned}$$

    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)

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

	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)

Outline

Bottom-up parsing

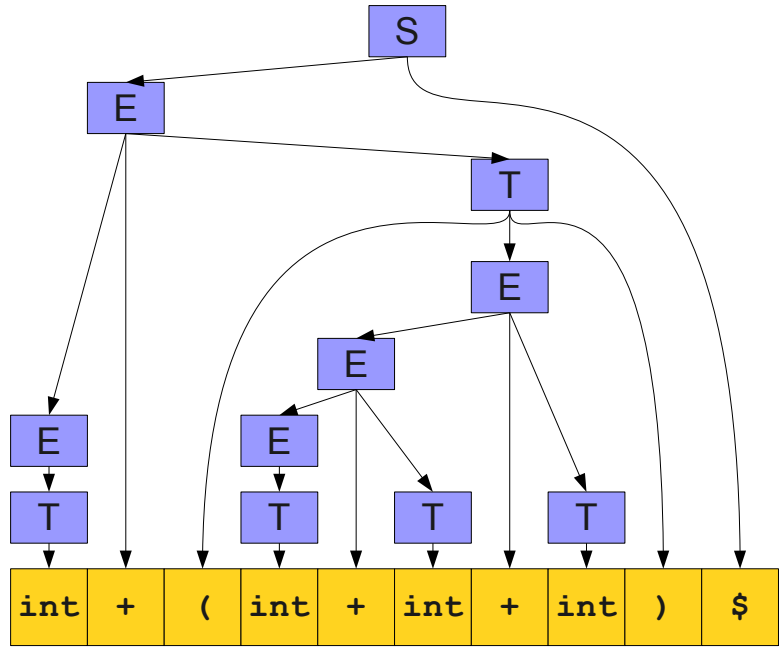
- A bottom-up parser creates the parse tree starting from the leaves towards the root
- It tries to convert the program into the start symbol
- Most common form of bottom-up parsing: shift-reduce parsing

Bottom-up parsing: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E\$ \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of
$int + (int + int + int)$



(Keith Schwarz)

Bottom-up parsing: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E\$ \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of $int + (int + int + int)$:

$int + (int + int + int)\$$
$T + (int + int + int)\$$
$E + (int + int + int)\$$
$E + (T + int + int)\$$
$E + (E + int + int)\$$
$E + (E + T + int)\$$
$E + (E + int)\$$
$E + (E + T)\$$
$E + (E)\$$
$E + T\$$
$E\$$
S

Top-down parsing is often done as a rightmost derivation in reverse (There is only one if the grammar is unambiguous).

Terminology

- A Rightmost (canonical) derivation is a derivation where the rightmost nonterminal is replaced at each step. A rightmost derivation from α to β is noted $\alpha \overset{*}{\Rightarrow}_{rm} \beta$.
- A reduction transforms uwv to uAv if $A \to w$ is a production
- α is a right sentential form if $S \overset{*}{\Rightarrow}_{rm} \alpha$.
- A handle of a right sentential form γ ($= \alpha\beta w$) is a production $A \to \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ:

$$S \overset{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$$

  - Informally, a handle is a production we can reverse without getting stuck.
  - If the handle is $A \to \beta$, we will also call β the handle.

Handle: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of $int + (int + int + int)$

$$
\begin{array}{l}
int + (int + int + int)\$ \\
T + (int + int + int)\$ \\
E + (int + int + int)\$ \\
E + (T + int + int)\$ \\
E + (E + int + int)\$ \\
E + (E + T + int)\$ \\
E + (E + int)\$ \\
E + (E + T)\$ \\
E + (E)\$ \\
E + T\$ \\
E\$ \\
S
\end{array}
$$

The handle is in red in each right sentential form

Finding the handles

- Bottom-up parsing = finding the handle in the right sentential form obtained at each step
- This handle is unique as soon as the grammar is unambiguous (because in this case, the rightmost derivation is unique)
- Suppose that our current form is uvw and the handle is $A \to v$ (getting uAw after reduction). w can not contain any nonterminals (otherwise we would have reduced a handle somewhere in w)

Shift/reduce parsing

Proposed model for a bottom-up parser:

- Split the input into two parts:
  - Left substring is our work area
  - Right substring is the input we have not yet processed
- All handles are reduced in the left substring
- Right substring consists only of terminals
- At each point, decide whether to:
  - Move a terminal across the split (shift)
  - Reduce a handle (reduce)

Shift/reduce parsing: example

Grammar:

$$
\begin{array}{lcl}
E & \rightarrow & E + T | T \\
T & \rightarrow & T * F | F \\
F & \rightarrow & (E) | \textbf{id}
\end{array}
$$

Bottum-up parsing of $id + id * id$

Left substring	Right substring	Action
\$	$id + id * id\$$	Shift
$\$id$	$+id * id\$$	Reduce by $F \rightarrow$ **id**
$\$F$	$+id * id\$$	Reduce by $T \rightarrow F$
$\$T$	$+id * id\$$	Reduce by $E \rightarrow T$
$\$E$	$+id * id\$$	Shift
$\$E+$	$id * id\$$	Shift
$\$E + id$	$*id\$$	Reduce by $F \rightarrow$ **id**
$\$E + F$	$*id\$$	Reduce by $T \rightarrow F$
$\$E + T$	$*id\$$	Shift
$\$E + T*$	$id\$$	Shift
$\$E + T * id$	\$	Reduce by $F \rightarrow$ **id**
$\$E + T * F$	\$	Reduce by $T \rightarrow T * F$
$\$E + T$	\$	Reduce by $E \rightarrow E + T$
$\$E$	\$	Accept

Shift/reduce parsing

- In the previous example, all the handles were to the far right end of the left area (not inside)
- This is convenient because we then never need to shift from the left to the right and thus could process the input from left-to-right in one pass.

- Is it the case for all grammars? Yes !
- Sketch of proof: by induction on the number of reduces
  - After no reduce, the first reduction can be done at the right end of the left area
  - After at least one reduce, the very right of the left area is a nonterminal (by induction hypothesis). This nonterminal must be part or at the left of the next handle, since we are tracing a rightmost derivation backwards.

Shift/reduce parsing

- Consequence: the left area can be represented by a stack (as all activities happen at its far right)

- Four possible actions of a shift-reduce parser:
  1. Shift: push the next terminal onto the stack
  2. Reduce: Replace the handle on the stack by the nonterminal
  3. Accept: parsing is successfully completed
  4. Error: discover a syntax error and call an error recovery routine

Shift/reduce parsing

- There still remain two open questions: At each step:
  - How to choose between shift and reduce?
  - If the decision is to reduce, which rules to choose (i.e., what is the handle)?
- Ideally, we would like this choice to be deterministic given the stack and the next k input symbols (to avoid backtracking), with k typically small (to make parsing efficient)
- Like for top-down parsing, this is not possible for all grammars
- Possible conflicts:
  - shift/reduce conflict: it is not possible to decide between shifting or reducing
  - reduce/reduce conflict: the parser can not decide which of several reductions to make

Shift/reduce parsing

We will see two main categories of shift-reduce parsers:

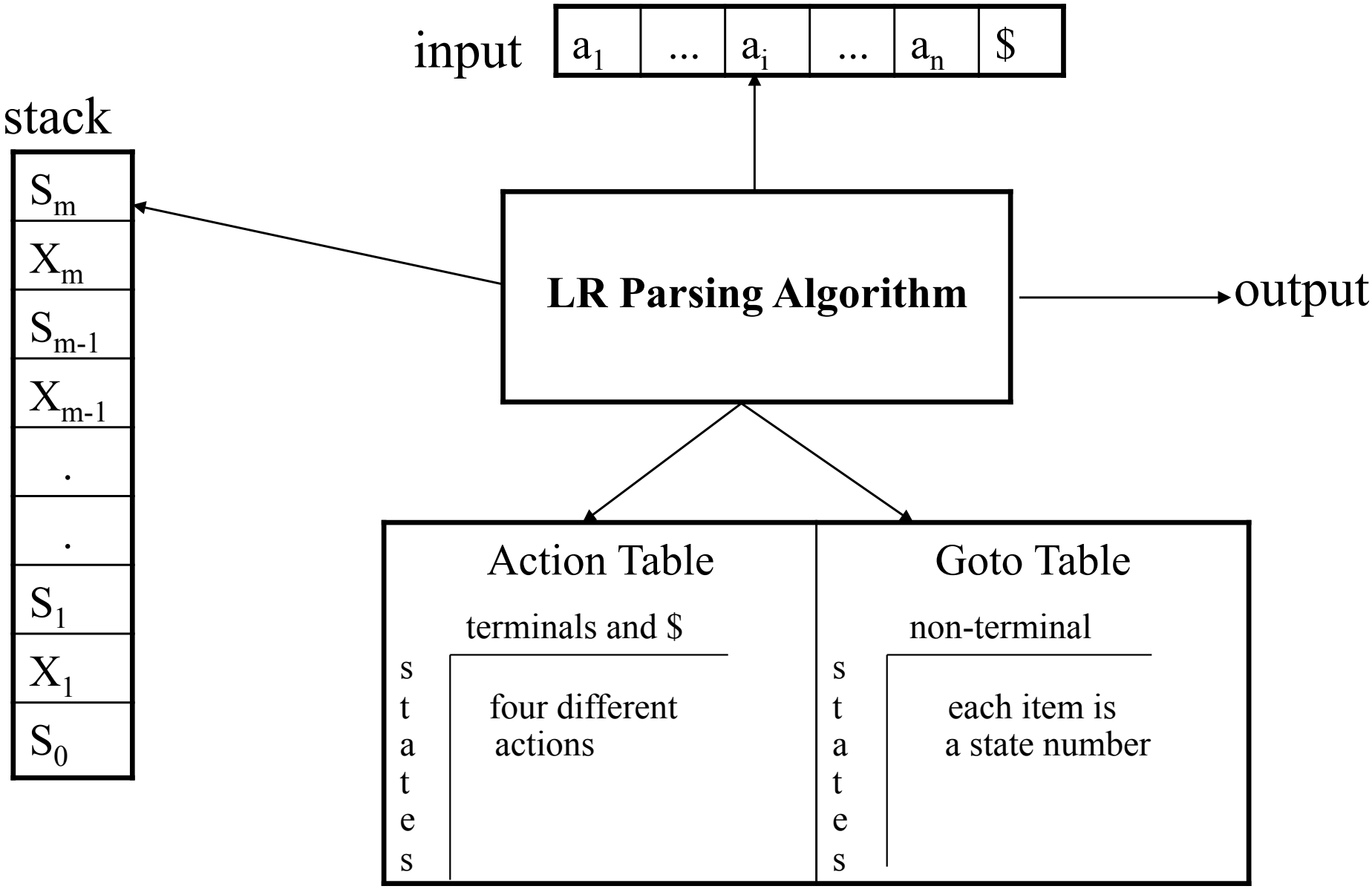
- LR-parsers
  - They cover a wide range of grammars
  - Different variants from the most specific to the most general: SLR, LALR, LR
- Weak precedence parsers
  - They work only for a small class of grammars
  - They are less efficient than LR-parsers
  - They are simpler to implement

Outline

LR-parsers

- LR(k) parsing: Left-to-right, Rightmost derivation, k symbols lookahead.
- Advantages:
  - The most general non-backtracking shift-reduce parsing, yet as efficient as other less general techniques
  - Can detect syntactic error as soon as possible (on a left-to-right scan of the input)
  - Can recognize virtually all programming language constructs (that can be represented by context-free grammars)
  - Grammars recognized by LR parsers is a proper superset of grammars recognized by predictive parsers (LL(k) $\subset$ LR(k))
- Drawbacks:
  - More complex to implement than predictive (or operator precedence) parsers
- Like table-driven predictive parsing, LR parsing is based on a parsing table.

Structure of a LR parser



Structure of a LR parser

- A configuration of a LR parser is described by the status of its stack and the part of the input not analysed (shifted) yet:

$$(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

  where X_i are (terminal or nonterminal) symbols, a_i are terminal symbols, and s_i are state numbers (of a DFA)
- A configuration corresponds to the right sentential form

$$X_1 \dots X_m a_i \dots a_n$$

- Analysis is based on two tables:
  - an action table that associates an action $ACTION[s, a]$ to each state s and nonterminal a.
  - a goto table that gives the next state $GOTO[s, A]$ from state s after a reduction to a nonterminal A

Actions of a LR-parser

- Let us assume the parser is in configuration

$$(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

(initially, the state is $(s_0, a_1 a_2 \dots a_n \$)$*, where* $a_1 \dots a_n$ *is the input word)*

- $\mathrm{ACTION}[s_m, a_i]$ can take four values:
  1. Shift s: shifts the next input symbol and then the state s on the stack $(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n) \rightarrow (s_0 X_1 s_1 \dots X_m s_m a_i s, a_{i+1} \dots a_n)$
  2. Reduce $A \rightarrow \beta$ (denoted by rn where n is a production number)
     - Pop $2|\beta|$ $(= r)$ items from the stack
     - Push A and s where $s = GOTO[s_{m-r}, A]$
       $(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n) \rightarrow$
       $(s_0 X_1 s_1 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n)$
     - Output the prediction $A \rightarrow \beta$
  3. Accept: parsing is successfully completed
  4. Error: parser detected an error (typically an empty entry in the action table).

LR-parsing algorithm

Create a stack with the start state s_0
a = GETNEXTTOKEN()
while (True)
    s = POP()
    if ($ACTION[s, a] =$ shift t)
        Push a and t onto the stack
        a = GETNEXTTOKEN()
    elseif ($ACTION[s, a] =$ reduce $A \to \beta$)
        Pop $2|\beta|$ elements off the stack
        Let state t now be the state on the top of the stack
        Push A onto the stack
        Push $GOTO[t, A]$ onto the stack
        Output $A \to \beta$
    elseif ($ACTION[s, a] =$ accept)
        break // Parsing is over
    else call error-recovery routine

Example: parsing table for the expression grammar

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to$ **id**

	Action Table						Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example: LR parsing with the expression grammar

stack	**input**	**action**	**output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing the parsing tables

- There are several ways of building the parsing tables, among which:
  - LR(0): no lookahead, works for only very few grammars
  - SLR: the simplest one with one symbol lookahead. Works with less grammars than the next ones
  - LR(1): very powerful but generate potentially very large tables
  - LALR(1): tradeoff between the other approaches in terms of power and simplicity
  - LR(k), k$>$ 1: exploit more lookahead symbols

- Main idea of all methods: build a DFA whose states keep track of where we are in the parsing

Parser generators

- LALR(1) is used in most parser generators like Yacc/Bison
- We will nevertheless only see SLR in details:
  - It's simpler.
  - LALR(1) is only minorly more expressive.
  - When a grammar is SLR, then the tables produced by SLR are identical to the ones produced by LALR(1).
  - Understanding of SLR principles is sufficient to understand how to handle a grammar rejected by LALR(1) parser generators (see later).

LR(0) item

- An LR(0) item (or item for short) of a grammar G is a production of G with a dot at some position of the body.
- Example: $A \to XYZ$ yields four items:

$$A \to .XYZ$$
$$A \to X.YZ$$
$$A \to XY.Z$$
$$A \to XYZ.$$

  ($A \to \epsilon$ generates one item $A \to .$)
- An item indicates how much of a production we have seen at a given point in the parsing process.
  - $A \to X.YZ$ means we have just seen on the input a string derivable from X (and we hope to get next YZ).
- Each state of the SLR parser will correspond to a set of LR(0) items
- A particular collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers

Construction of the canonical LR(0) collection

- The grammar G is first augmented into a grammar G' with a new start symbol S' and a production $S' \to S$ where S is the start symbol of G
- We need to define two functions:
  - $\textsc{Closure}(I)$: extends the set of items I when some of them have a dot to the left of a nonterminal
  - $\textsc{Goto}(I, X)$: moves the dot past the symbol X in all items in I
- These two functions will help define a DFA:
  - whose states are (closed) sets of items
  - whose transitions (on terminal and nonterminal symbols) are defined by the $\textsc{Goto}$ function

Closure

Closure(I)

repeat

for any item $A \to \alpha.X\beta$ in I

for any production $X \to \gamma$

$I = I \cup \{X \to .\gamma\}$

until I does not change

return I

Example:

$E' \to E$
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

$$\text{Closure}(\{E' \to .E\}) = \{E' \to .E,$$
$$E \to .E + T$$
$$E \to .T$$
$$T \to .T * F$$
$$T \to .F$$
$$F \to .(E)$$
$$F \to .\ \textbf{id}\ \}$$

Goto

$\textsc{Goto}(I, X)$

Set J to the empty set
for any item $A \to \alpha.X\beta$ in I
$J = J \bigcup \{A \to \alpha X.\beta\}$
return $\textsc{closure}(J)$

Example:

$E' \to E$
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

$I_0 = \{E' \to .E,$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .$ **id** $\}$

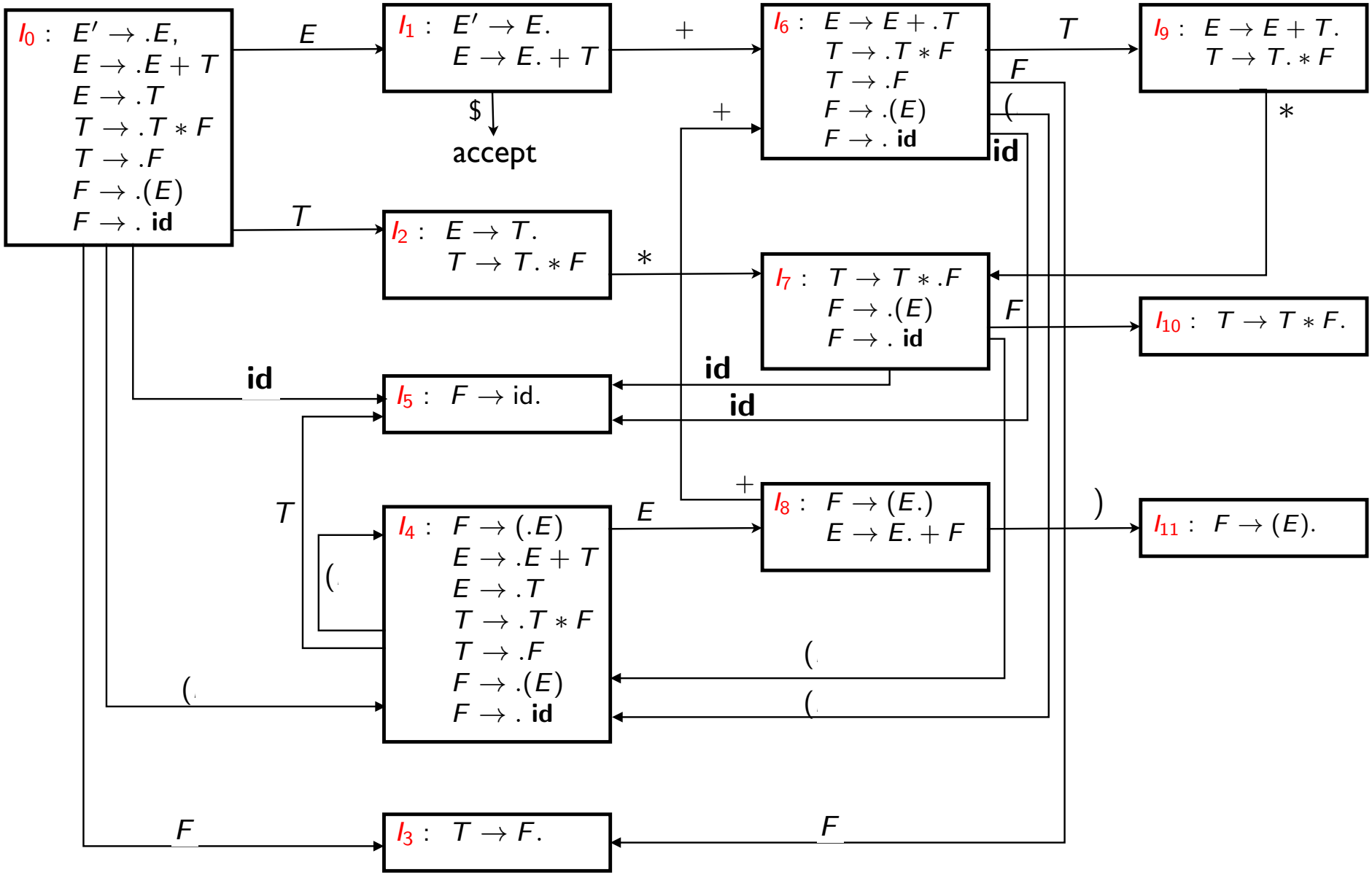
$\textsc{goto}(I_0, E) = \{E' \to E., E \to E. + T\}$
$\textsc{goto}(I_0, T) = \{E \to T., T \to T. * F\}$
$\textsc{goto}(I_0, F) = \{T \to F.\}$
$\textsc{goto}(I_0,'(') = \textsc{Closure}(\{F \to (.E)\})$
$= \{F \to (.E)\} \cup (I_0 \setminus \{E' \to E\})$
$\textsc{goto}(I_0, \mathsf{id}) = \{F \to \mathsf{id}.\}$

Construction of the canonical collection

$C = \{\textsc{closure}(\{S' \to .S\})\}$
repeat
  for each item set I in C
    for each item $A \to \alpha.X\beta$ in I
      $C = C \cup \{\textsc{Goto}(I, X)\}$
until C did not change in this iteration
return C

- Collect all sets of items reachable from the initial state by one or several applications of GOTO.
- Item sets in C are the states of a DFA, GOTO is its transition function

Example


I_0 : $E' \to .E$,
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .$ **id**
I_1 : $E' \to E.$
$E \to E. + T$
I_2 : $E \to T.$
$T \to T. * F$
I_3 : $T \to F.$
I_4 : $F \to (.E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .$ **id**
I_5 : $F \to$ id.
I_6 : $E \to E + .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .$ **id**
I_7 : $T \to T * .F$
$F \to .(E)$
$F \to .$ **id**
I_8 : $F \to (E.)$
$E \to E. + F$
I_9 : $E \to E + T.$
$T \to T. * F$
I_{10} : $T \to T * F.$
I_{11} : $F \to (E).$
E
T
F
+
*
(
)
id
$
accept

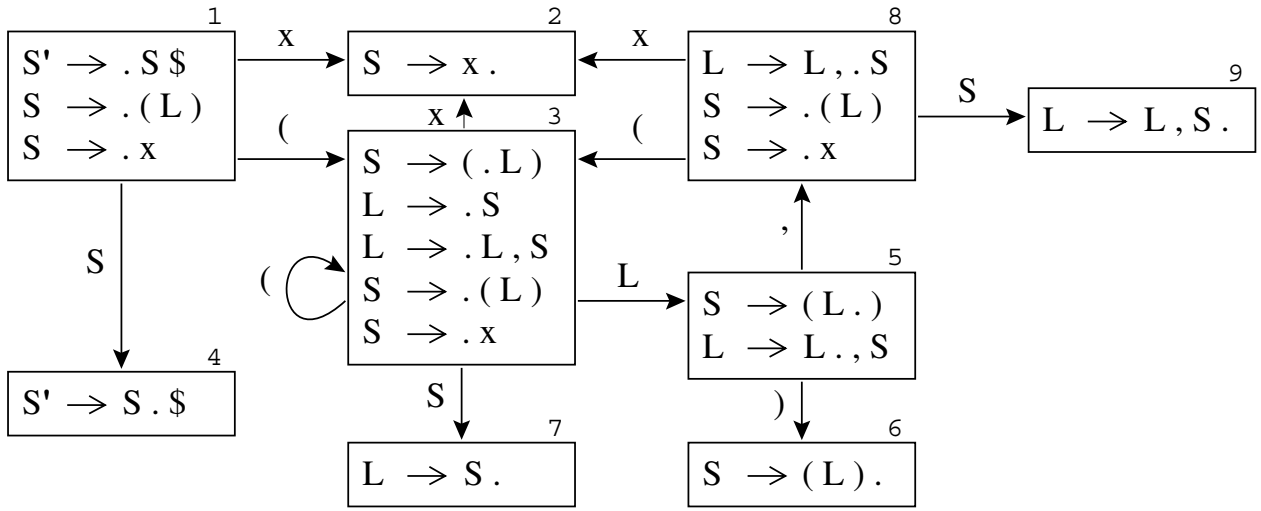
Constructing the LR(0) parsing table

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of $LR(0)$ items for G' (the augmented grammar)
2. State i of the parser is derived from I_i. Actions for state i are as follows:
   - 2.1 If $A \to \alpha.a\beta$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] = \text{Shift } j$
   - 2.2 If $A \to \alpha.$ is in I_i, then set $ACTION[i, a] = \text{Reduce } A \to \alpha$ for all terminals a.
   - 2.3 If $S' \to S.$ is in I_i, then set $ACTION[i, \$] = \text{Accept}$
3. If $\text{GOTO}(I_i, X) = I_j$, then $GOTO[i, X] = j$.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state s_0 is the set of items containing $S' \to .S$

$\Rightarrow$ *LR(0) because the chosen action (shift or reduce) only depends on the current state (but the choice of the next state still depends on the token)*

Example of a LR(0) grammar

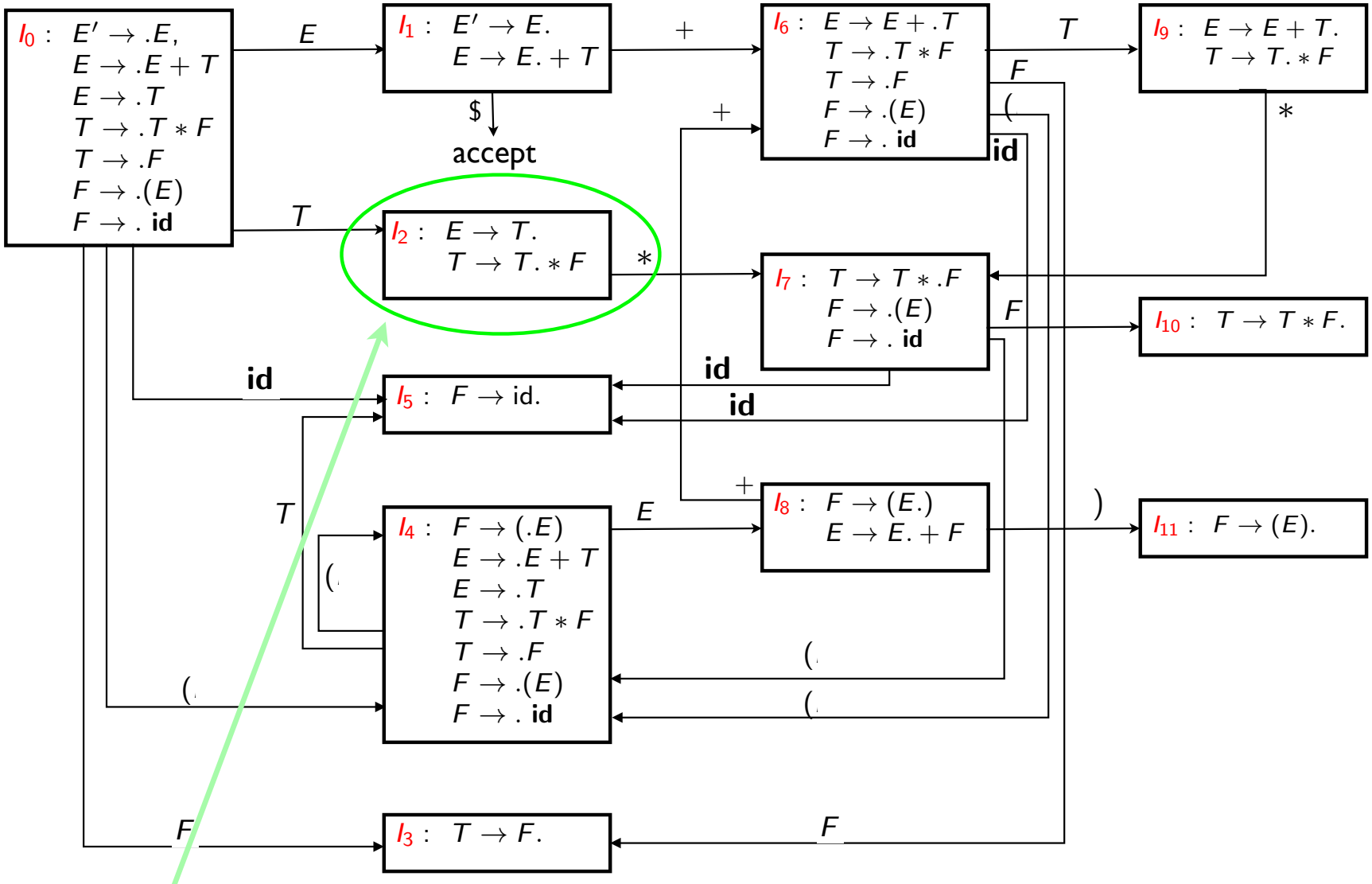
0 $S' \to S\$$
1 $S \to (L)$
2 $S \to x$
3 $L \to S$
4 $L \to L , S$



	(	)	x	,	$	*S*	*L*
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

(Appel)

Example of a non LR(0) grammar



Conflict: in state 2, we don't know whether to shift or reduce.

Constructing the SLR parsing tables

1. Construct $c = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of $LR(0)$ items for G' (the augmented grammar)
2. State i of the parser is derived from I_i. Actions for state i are as follows:
   - 2.1 If $A \to \alpha.a\beta$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] =$ Shift j
   - 2.2 If $A \to \alpha.$ is in I_i, then $ACTION[i, a] =$ Reduce $A \to \alpha$ for all terminals a in $Follow(A)$ where $A \neq S'$
   - 2.3 If $S' \to S.$ is in I_i, then set $ACTION[i, \$] =$ Accept
3. If $\text{GOTO}(I_i, A) = I_j$ for a nonterminal A, then $GOTO[i, A] = j$
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state s_0 is the set of items containing $S' \to .S$

$\Rightarrow$ *the simplest form of one symbol lookahead, SLR (Simple LR)*

Example

	First	*Follow*
E	**id** (	$ +)
T	**id** (	$ + *)
F	**id** (	$ + *)

	Action Table							Goto Table		
state	**id**	+	*	(	)	**$**		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

SLR(1) grammars

- A grammar for which there is no (shift/reduce or reduce/reduce) conflict during the construction of the SLR table is called SLR(1) (or SLR in short).
- All SLR grammars are unambiguous but many unambiguous grammars are not SLR
- There are more SLR grammars than LL(1) grammars but there are LL(1) grammars that are not SLR.

Conflict example for SLR parsing

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

$$
\begin{array}{ll}
I_0: & S' \rightarrow \cdot S \\
& S \rightarrow \cdot L = R \\
& S \rightarrow \cdot R \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id} \\
& R \rightarrow \cdot L \\
\\
I_1: & S' \rightarrow S\cdot \\
\\
I_2: & S \rightarrow L\cdot = R \\
& R \rightarrow L\cdot \\
\\
I_3: & S \rightarrow R\cdot \\
\\
I_4: & L \rightarrow *\cdot R \\
& R \rightarrow \cdot L \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id}
\end{array}
$$

$$
\begin{array}{ll}
I_5: & L \rightarrow \mathbf{id}\cdot \\
\\
I_6: & S \rightarrow L = \cdot R \\
& R \rightarrow \cdot L \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id} \\
\\
I_7: & L \rightarrow *R\cdot \\
\\
I_8: & R \rightarrow L\cdot \\
\\
I_9: & S \rightarrow L = R\cdot
\end{array}
$$

(Dragonbook)

Follow(R) contains '='. In I_2, when seeing '=' on the input, we don't know whether to shift or to reduce with $R \rightarrow L$.

Summary of SLR parsing

Construction of a SLR parser from a CFG grammar

- Eliminate ambiguity *(or not, see later)*
- Add the production $S' \to S$, where S is the start symbol of the grammar
- Compute the LR(0) canonical collection of LR(0) item sets and the GOTO function (transition function)
- Add a shift action in the action table for transitions on terminals and goto actions in the goto table for transitions on nonterminals
- Compute *Follow* for each nonterminals (which implies first adding $S'' \to S'\$$ to the grammar and computing *First* and *Nullable*)
- Add the reduce actions in the action table according to *Follow*
- Check that the grammar is SLR *(and if not, try to resolve conflicts, see later)*

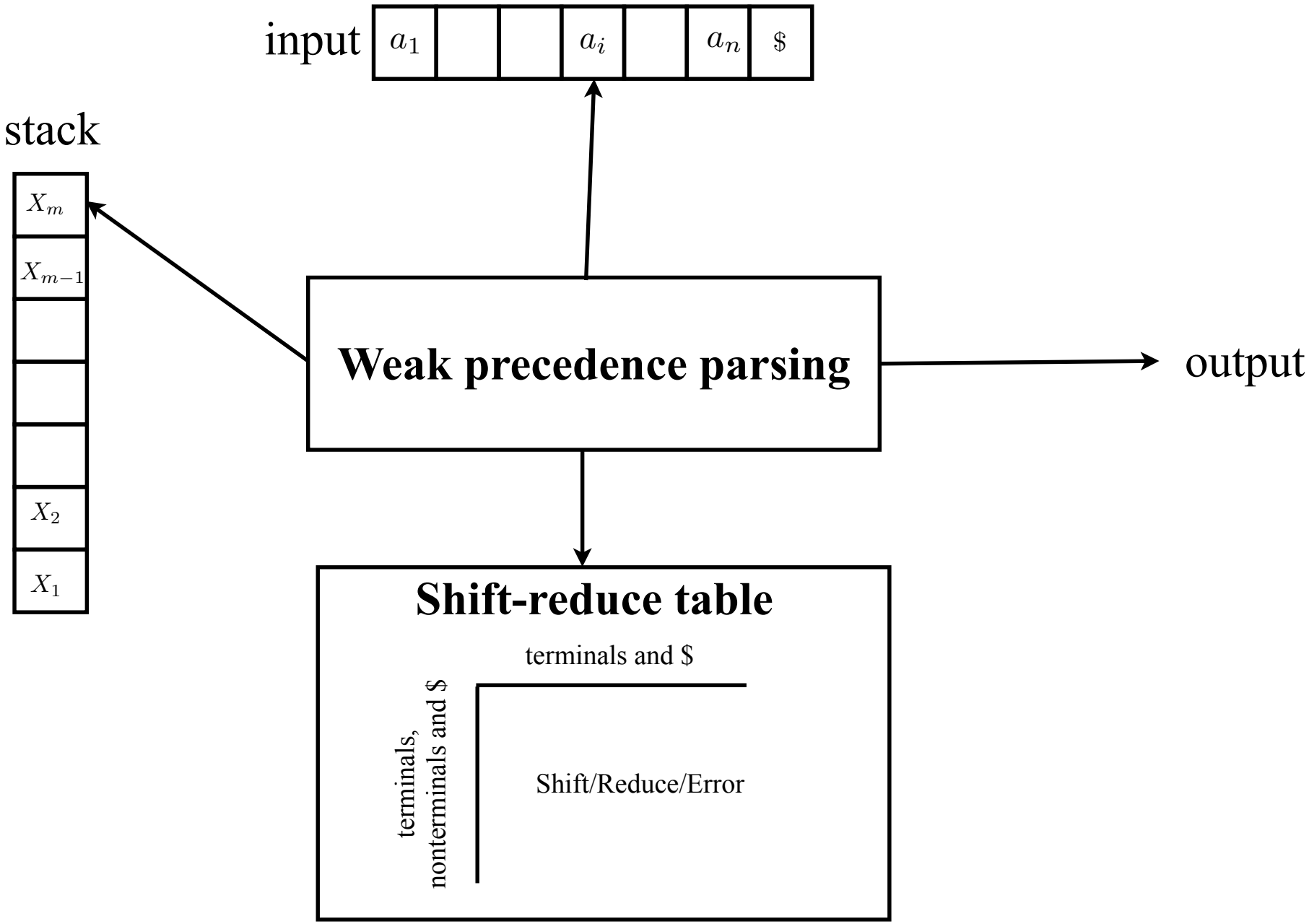
Outline

Operator precedence parsing

- Bottom-up parsing methods that follow the idea of shift-reduce parsers
- Several flavors: operator, simple, and weak precedence.
- In this course, only weak precedence

- Main differences compared to LR parsers:
  - There is no explicit state associated to the parser (and thus no state pushed on the stack)
  - The decision of whether to shift or reduce is taken based solely on the symbol on the top of the stack and the next input symbol (and stored in a shift-reduce table)
  - In case of reduction, the handle is the longest sequence at the top of stack matching the RHS of a rule

Structure of the weak precedence parser



Weak precedence parsing algorithm

```
Create a stack with the special symbol $
a = GETNEXTTOKEN()
while (True)
    if (Stack== $S and a == $)
        break // Parsing is over
    X_m = TOP(Stack)
    if (SRT[X_m, a] = shift)
        Push a onto the stack
        a = GETNEXTTOKEN()
    elseif (SRT[X_m, a] = reduce)
        Search for the longest RHS that matches the top of the stack
        if no match found
            call error-recovery routine
        Let denote this rule by Y → X_{m-r+1} ... X_m
        Pop r elements off the stack
        Push Y onto the stack
        Output Y → X_{m-r+1} ... X_m
    else call error-recovery routine
```

Example for the expression grammar

Example:

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Shift/reduce table

	$*$	$+$	(	)	**id**	$
E		S		S		R
T	S	R		R		R
F	R	R		R		R
$*$			S		S	
$+$			S		S	
(			S		S	
)	R	R		R		R
id	R	R		R		R
$			S		S	

Example of parsing

Stack	Input	Action
$\$$	$id + id * id\$$	Shift
$\$id$	$+id * id\$$	Reduce by $F \rightarrow$ **id**
$\$F$	$+id * id\$$	Reduce by $T \rightarrow F$
$\$T$	$+id * id\$$	Reduce by $E \rightarrow T$
$\$E$	$+id * id\$$	Shift
$\$E+$	$id * id\$$	Shift
$\$E + id$	$*id\$$	Reduce by $F \rightarrow$ **id**
$\$E + F$	$*id\$$	Reduce by $T \rightarrow F$
$\$E + T$	$*id\$$	Shift
$\$E + T*$	$id\$$	Shift
$\$E + T * id$	$\$$	Reduce by $F \rightarrow$ **id**
$\$E + T * F$	$\$$	Reduce by $T \rightarrow T * F$
$\$E + T$	$\$$	Reduce by $E \rightarrow E + T$
$\$E$	$\$$	Accept

Precedence relation: principle

- We define the (weak precedence) relations $\lessdot$ and $\gtrdot$ between symbols of the grammar (terminals or nonterminals)
  - $X \lessdot Y$ if XY appears in the RHS of a rule or if X precedes a reducible word whose leftmost symbol is Y
  - $X \gtrdot Y$ if X is the rightmost symbol of a reducible word and Y the symbol immediately following that word
- Shift when $X_m \lessdot a$, reduce when $X_m \gtrdot a$
- Reducing changes the precedence relation only at the top of the stack (there is thus no need to shift backward)

Precedence relation: formal definition

- Let $G = (V, \Sigma, R, S)$ be a context-free grammar and \$ a new symbol acting as left and right end-marker for the input word. Define $V' = V \cup \{\$\}$
- The weak precedence relations $\lessdot$ and $\gtrdot$ are defined respectively on $V' \times V$ and $V \times V'$ as follows:
  1. $X \lessdot Y$ if $A \to \alpha X B \beta$ is in R, and $B \stackrel{+}{\Rightarrow} Y\gamma$,
  2. $X \lessdot Y$ if $A \to \alpha X Y \beta$ is in R
  3. $\$ \lessdot X$ if $S \stackrel{+}{\Rightarrow} X\alpha$
  4. $X \gtrdot a$ if $A \to \alpha B \beta$ is in R, and $B \stackrel{+}{\Rightarrow} \gamma X$ and $\beta \stackrel{*}{\Rightarrow} a\gamma$
  5. $X \gtrdot \$$ if $S \stackrel{+}{\Rightarrow} \alpha X$

  for some α, β, γ, and B

Construction of the SR table: shift

Shift relation, $\lessdot$:

Initialize $\mathcal{S}$ to the empty set.
1 add $\$ \lessdot S$ to $\mathcal{S}$
2 **for** each production $X \to L_1 L_2 \ldots L_k$
  for $i = 1$ **to** $k - 1$
    add $L_i \lessdot L_{i+1}$ to $\mathcal{S}$
3 **repeat**
  for each* pair $X \lessdot Y$ in $\mathcal{S}$
    for each production $Y \to L_1 L_2 \ldots L_k$
      Add $X \lessdot L_1$ to $\mathcal{S}$
until $\mathcal{S}$ did not change in this iteration.

* We only need to consider the pairs $X \lessdot Y$ with Y a nonterminal that were added in $\mathcal{S}$ at the previous iteration

Example of the expression grammar: shift

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Step 1	$S \lessdot \$$
Step 2	$E \lessdot +$
	$+ \lessdot T$
	$T \lessdot *$
	$* \lessdot F$
	$(\lessdot E$
	$E \lessdot)$
Step 3.1	$+ \lessdot F$
	$* \lessdot$ **id**
	$* \lessdot ($
	$(\lessdot T$
Step 3.2	$+ \lessdot$ **id**
	$+ \lessdot ($
	$(\lessdot F$
Step 3.3	$(\lessdot ($
	$(\lessdot$ **id**

Construction of the SR table: reduce

Reduce relation, $\gtrdot$:

Initialize $\mathcal{R}$ to the empty set.
1 add $S \gtrdot \$$ to $\mathcal{R}$
2 **for** each production $X \to L_1 L_2 \ldots L_k$
    for each pair $X \lessdot Y$ in $\mathcal{S}$
        add $L_k \gtrdot Y$ in $\mathcal{R}$
3 **repeat**
    for each* pair $X \gtrdot Y$ in $\mathcal{R}$
        for each production $X \to L_1 L_2 \ldots L_k$
            Add $L_k \gtrdot Y$ to $\mathcal{R}$
until $\mathcal{R}$ did not change in this iteration.

* We only need to consider the pairs $X \gtrdot Y$ with X a nonterminal that were added in $\mathcal{R}$ at the previous iteration.

Example of the expression grammar: reduce

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Step 1	$E \gtrdot \$$
Step 2	$T \gtrdot +$
	$F \gtrdot *$
	$T \gtrdot)$
Step 3.1	$T \gtrdot \$$
	$F \gtrdot +$
	$) \gtrdot *$
	id $\gtrdot *$
	$F \gtrdot)$
Step 3.2	$F \gtrdot \$$
	$) \gtrdot +$
	id $\gtrdot +$
	$) \gtrdot)$
	id $\gtrdot)$
Step 3.3	**id** $\gtrdot \$$
	$) \gtrdot \$$

Weak precedence grammars

- Weak precedence grammars are those that can be analysed by a weak precedence parser.
- A grammar $G = (V, \Sigma, R, S)$ is called a weak precedence grammar if it satisfies the following conditions:
  1. There exist no pair of productions with the same right hand side
  2. There are no empty right hand sides ($A \to \epsilon$)
  3. There is at most one weak precedence relation between any two symbols
  4. Whenever there are two syntactic rules of the form $A \to \alpha X \beta$ and $B \to \beta$, we don't have $X \lessdot B$
- Conditions 1 and 2 are easy to check
- Conditions 3 and 4 can be checked by constructing the SR table.

Example of the expression grammar

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Shift/reduce table

	$*$	$+$	(	)	**id**	$
E		S		S		R
T	S	R		R		R
F	R	R		R		R
$*$			S		S	
$+$			S		S	
(			S		S	
)	R	R		R		R
id	R	R		R		R
$			S		S	

- Conditions 1-3 are satisfied (there is no conflict in the SR table)
- Condition 4:
  - $E \to E + T$ and $E \to T$ but we don't have $+ \lessdot E$ (see slide 202)
  - $T \to T * F$ and $T \to F$ but we don't have $* \lessdot T$ (see slide 202)

Removing ϵ rules

- Removing rules of the form $A \to \epsilon$ is not difficult
- For each rule with A in the RHS, add a set of new rules consisting of the different combinations of A replaced or not with ϵ.
- Example:

$$\begin{aligned} S &\to AbA|B \\ B &\to b|c \\ A &\to \epsilon \end{aligned}$$

is transformed into

$$\begin{aligned} S &\to AbA|Ab|bA|b|B \\ B &\to b|c \end{aligned}$$

Summary of weak precedence parsing

Construction of a weak precedence parser

- Eliminate ambiguity *(or not, see later)*
- Eliminate productions with ϵ and ensure that there are no two productions with identical RHS
- Construct the shift/reduce table
- Check that there is no conflict during the construction
- Check condition 4 of slide 205

Outline

Using ambiguous grammars with bottom-up parsers

- All grammars used in the construction of Shift/Reduce parsing tables must be un-ambiguous
- We can still create a parsing table for an ambiguous grammar but there will be conflicts
- We can often resolve these conflicts in favor of one of the choices to disambiguate the grammar
- Why use an ambiguous grammar?
  - Because the ambiguous grammar is much more natural and the corresponding unambiguous one can be very complex
  - Using an ambiguous grammar may eliminate unnecessary reductions
- Example:

$$E \to E + E | E * E | (E) | \mathbf{id} \quad \Rightarrow \quad \begin{array}{l} E \to E + T | T \\ T \to T * F | F \\ F \to (E) | \mathbf{id} \end{array}$$

Set of LR(0) items of the ambiguous expression grammar

$E \to E + E | E * E | (E) | \mathbf{id}$

$Follow(E) = \{\$, +, *,)\}$
$\Rightarrow$ states 7 and 8 have shift/reduce conflicts for $+$ and $*$.

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id} \cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

(Dragonbook)

Disambiguation

Example:

- Parsing of $\mathbf{id} + \mathbf{id} * \mathbf{id}$ will give the configuration

$$(0E1 + 4E7, *\mathbf{id}\$)$$

We can choose:

- $ACTION[7, *]$ =shift 5$\Rightarrow$ precedence to $*$
- $ACTION[7, *]$ =reduce $E \to E + E \Rightarrow$ precedence to $+$

- Parsing of $\mathbf{id} + \mathbf{id} + \mathbf{id}$ will give the configuration

$$(0E1 + 4E7, +\mathbf{id}\$)$$

We can choose:

- $ACTION[7, +]$ =shift 4$\Rightarrow$ $+$ is right-associative
- $ACTION[7, +]$ =reduce $E \to E + E \Rightarrow$ $+$ is left-associative

(same analysis for I_8)

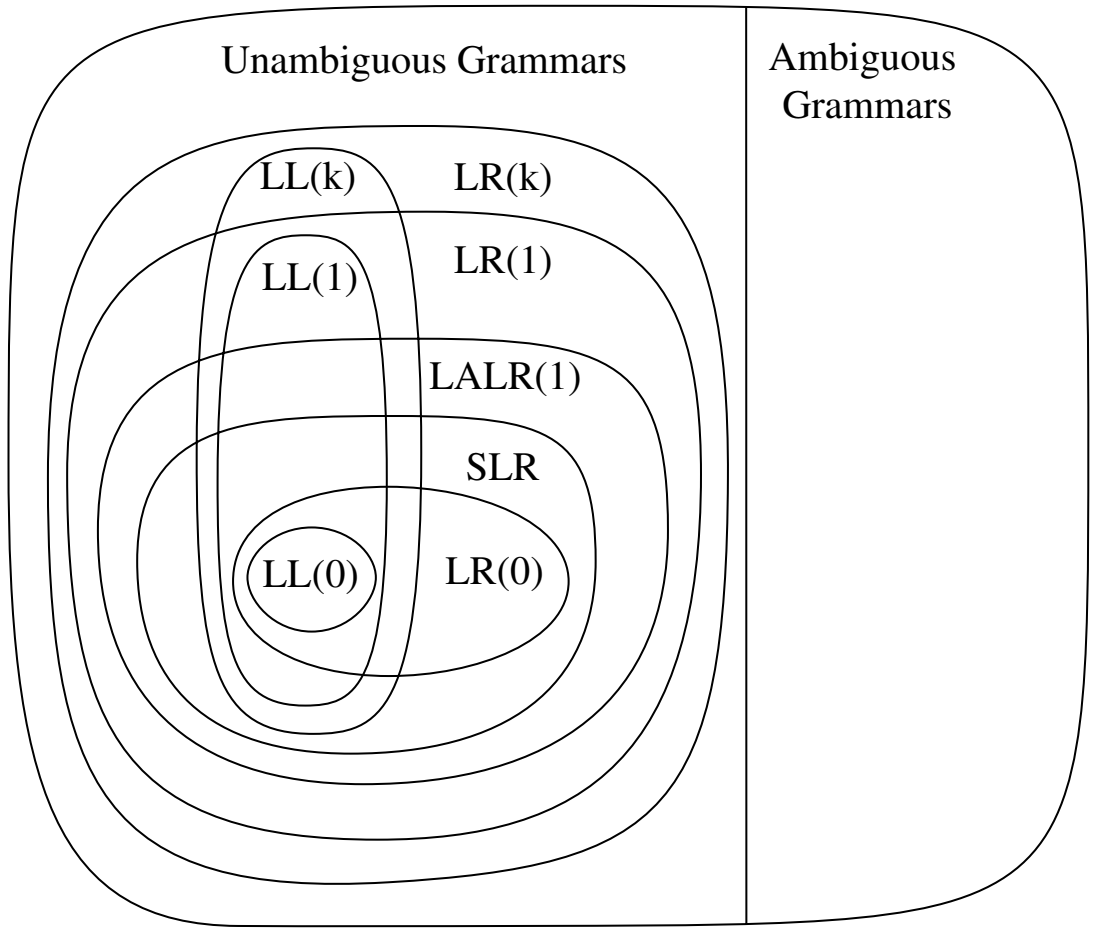
outline

1. Introduction
2. Context-free grammar
3. Top-down parsing
4. Bottom-up parsing
    - Shift/reduce parsing
    - LR parsers
    - Operator precedence parsing
    - Using ambiguous grammars
5. Conclusion and some practical considerations

Top-down versus bottom-up parsing

- Top-down
  - Easier to implement (recursively), enough for most standard programming languages
  - Need to modify the grammar sometimes strongly, less general than bottom-up parsers
  - Used in most hand-written compilers and some parser generators (JavaCC, ANTLR)
- Bottom-up:
  - More general, less strict rules on the grammar, SLR(1) powerful enough for most standard programming languages
  - More difficult to implement, less easy to maintain (add new rules, etc.)
  - Used in most parser generators (Yacc, Bison)

Hierarchy of grammar classes



(Appel)

Error detection and recovery

- In table-driven parsers, there is an error as soon as the table contains no entry (or an error entry) for the current stack (state) and input symbols
- The least one can do: report a syntax error and give information about the position in the input file and the tokens that were expected at that position
- In practice, it is however desirable to continue parsing to report more errors
- There are several ways to recover from an error:
  - Panic mode
  - Phrase-level recovery
  - Introduce specific productions for errors
  - Global error repair

Panic-mode recovery

- In case of syntax error within a "phrase", skip until the next synchronizing token is found (e.g., semicolon, right parenthesis) and then resume parsing
- In LR parsing:
  - Scan down the stack until a state s with a goto on a particular nonterminal A is found
  - Discard zero or more input symbols until a symbol a is found that can follow A
  - Stack the state $GOTO(s, A)$ and resume normal parsing

Phrase-level recovery

- Examine each error entry in the parsing table and decide on an appropriate recovery procedure based on the most likely programmer error.
- Examples in LR parsing: $E \rightarrow E + E|E * E|(E)|id$
  - $id + *id$:
    $*$ is unexpected after a $+$: report a "missing operand" error, push an arbitrary number on the stack and go to the appropriate next state
  - $id + id) + id$:
    Report an "unbalanced right parenthesis" error and remove the right parenthesis from the input

Other error recovery approaches

Introduce specific productions for detecting errors:

- Add rules in the grammar to detect common errors
- Examples for a *C* compiler:
  $I \rightarrow$ **if** E I (parenthesis are missing around the expression)
  $I \rightarrow$ **if** (E) **then** I (**then** is not needed in C)

Global error repair:

- Try to find *globally* the smallest set of insertions and deletions that would turn the program into a syntactically correct string
- Very costly and not always effective

Building the syntax tree

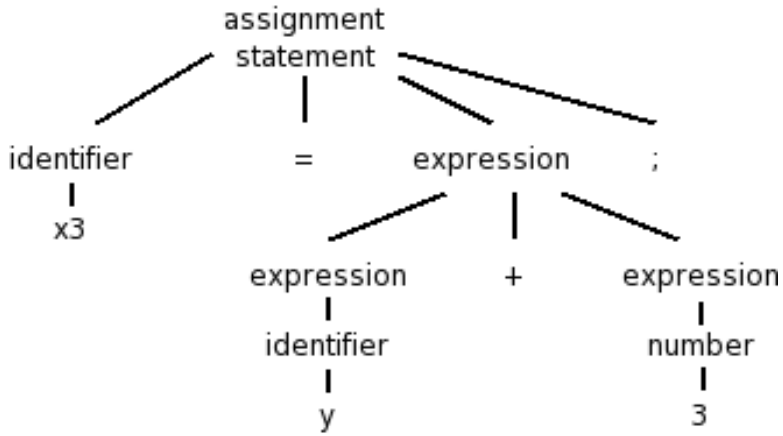
- Parsing algorithms presented so far only check that the program is syntactically correct
- In practice, the parser also needs to build the parse tree (also called concrete syntax tree)
- Its construction is easily embedded into the parsing algorithm

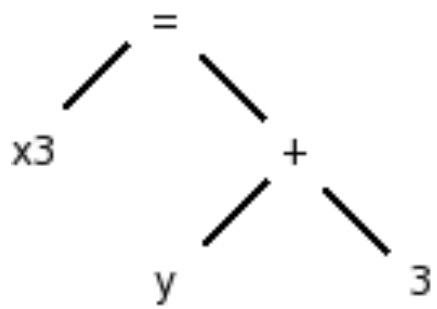
- Top-down parsing:
  - Recursive descent: let each parsing function return the sub-trees for the parts of the input they parse
  - Table-driven: each nonterminal on the stack points to its node in the partially built syntax tree. When the nonterminal is replaced by one of its RHS, nodes for the symbols on the RHS are added as children to the nonterminal node

Building the syntax tree

- Bottom-up parsing:
  - Each stack element points to a subtree of the syntax tree
  - When performing a reduce, a new syntax tree is built with the nonterminal at the root and the popped-off stack elements as children

- Note:
  - In practice, the concrete syntax tree is not built but rather a simplified (abstract) syntax tree
  - Depending on the complexity of the compiler, the syntax tree might even not be constructed





For your project

- The choice of a parsing technique is left open for the project
- You can either use a parser generator or implement the parser by yourself
- Motivate your choice in your report and explain any transformation you had to apply to your grammar to make it fit the constraints of the parser

- Parser generators:
  - Yacc: Unix parser generator, LALR(1) (companion of Lex)
  - Bison: free implementation of Yacc, LALR(1) (companion of Flex)
  - ANTLR: LL(*), implemented in Java but output code in several languages
  - ...
- `http://en.wikipedia.org/wiki/Comparison_of_parser_generators`

An example with Flex/Bison

Example: Parsing of the following expression grammar:

$$
\begin{aligned}
\mathit{Input} &\rightarrow \mathit{Input}\ \mathit{Line} \\
\mathit{Input} &\rightarrow \epsilon \\
\mathit{Line} &\rightarrow \mathit{Exp}\ \mathit{EOL} \\
\mathit{Line} &\rightarrow \mathit{EOL} \\
\mathit{Exp} &\rightarrow \textbf{num} \\
\mathit{Exp} &\rightarrow \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp} - \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp} * \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp}/\mathit{Exp} \\
\mathit{Exp} &\rightarrow (\mathit{Exp})
\end{aligned}
$$

`https://github.com/prashants/calc`

Flex file: calc.lex

```
%{
#define YYSTYPE double /* Define the main semantic type */
#include "calc.tab.h"  /* Define the token constants    */
#include <stdlib.h>
%}
%option yylineno       /* Ask flex to put line number in yylineno */
white [ \t]+
digit [0-9]
integer {digit}+
exponent [eE][+-]?{integer}
real {integer}("."{integer})?{exponent}?
%%
{white} {}
{real} { yylval=atof(yytext); return NUMBER; }
"+" { return PLUS; }
"-" { return MINUS; }
"*" { return TIMES; }
"/" { return DIVIDE; }
"(" { return LEFT; }
")" { return RIGHT; }
"\n" { return END; }
. { yyerror("Invalid token"); }
```

Bison file: calc.y

- Declaration:

```
%{
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define YYSTYPE double  /* Define the main semantic type */
extern char *yytext;    /* Global variables of Flex */
extern int yylineno;
extern FILE *yyin;
%}
```

- Definition of the tokens and start symbol

```
%token NUMBER
%token PLUS MINUS TIMES DIVIDE
%token LEFT RIGHT
%token END

%start Input
```

Bison file: calc.y

- Operator associativity and precedence:

```
%left PLUS MINUS
%left TIMES DIVIDE
%left NEG
```

- Production rules and associated actions:

```
%%

Input:                    /* epsilon */
        | Input Line
;

Line:
        END
        | Expression END { printf("Result: %f\n", $1); }
;
```

Bison file: calc.y

- Production rules and actions (continued):

```
Expression:
        NUMBER { $$ = $1; }
        | Expression PLUS Expression { $$ = $1 + $3; }
        | Expression MINUS Expression { $$ = $1 - $3; }
        | Expression TIMES Expression { $$ = $1 * $3; }
        | Expression DIVIDE Expression { $$ = $1 / $3; }
        | MINUS Expression %prec NEG { $$ = -$2; }
        | LEFT Expression RIGHT { $$ = $2; }
;
```

- Error handling:

```
%%

int yyerror(char *s)
{
        printf("%s on line %d - %s\n", s, yylineno, yytext);
}
```

Bison file: calc.y

- Main functions:

```
int main(int argc, char **argv)
{
   /* if any input file has been specified read from that */
   if (argc >= 2) {
      yyin = fopen(argv[1], "r");
      if (!yyin) {
         fprintf(stderr, "Failed to open input file\n");
      }
      return EXIT_FAILURE;
   }

   if (yyparse()) {
      fprintf(stdout, "Successful parsing\n");
   }

   fclose(yyin);
   fprintf(stdout, "End of processing\n");
   return EXIT_SUCCESS;
}
```

Bison file: makefile

- How to compile:

```
bison -v -d calc.y
flex -o calc.lex.c calc.lex
gcc -o calc calc.lex.c calc.tab.c -lfl -lm
```

- Example:

```
>./calc
1+2*3-4
Result: 3.000000
1+3*-4
Result: -11.000000
*2
syntax error on line 3 - *
Successful parsing
End of processing
```

The state machine

Excerpt of calc.output (with *Expression* abbreviated in *Exp*):

```
state 9
    6 Exp: Exp . PLUS Exp
    7    | Exp . MINUS Exp
    8    | Exp . TIMES Exp
    9    | Exp . DIVIDE Exp
   10    | MINUS Exp .

    $default  reduce using rule 10 (Exp)

state 10
    6 Exp: Exp . PLUS Exp
    7    | Exp . MINUS Exp
    8    | Exp . TIMES Exp
    9    | Exp . DIVIDE Exp
   11    | LEFT Exp . RIGHT

    PLUS    shift, and go to state 11
    MINUS   shift, and go to state 12
    TIMES   shift, and go to state 13
    DIVIDE  shift, and go to state 14
    RIGHT   shift, and go to state 16
```

```
state 11
    6 Exp: Exp PLUS . Exp

    NUMBER  shift, and go to state 3
    MINUS   shift, and go to state 4
    LEFT    shift, and go to state 5

    Exp  go to state 17
```