3. Syntax Analysis

Andrea Polini, Luca Tesei

Formal Languages and Compilers
MSc in Computer Science
University of Camerino

ToC

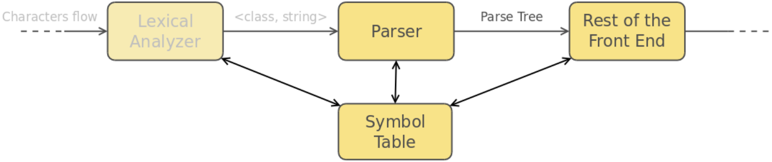
Syntax analysis

Parsing

Parsing is the activity of taking a string of terminals and figuring out how to derive it from the start symbol of a grammar. If a derivation cannot be obtained then syntax errors must be reported within the string.

The Parser

The parser obtains a sequence of tokens and verifies that the sequence can be correctly generated by a given grammar of the source language. For well-formed programs the parser will generate a parse tree that will be passed to the next compiler phase.



Parse Tree

Parse tree

A parse tree shows how the start symbol of a grammar derives the string in the language. If $A \rightarrow XYZ$ is a production applied in a derivation, the parse tree will have an interior node labeled with A with three children labeled X, Y, Z from left to right:

- the root is always labeled with the start symbols
- leaves are labeled with terminals or ϵ
- interior nodes are labeled with non-terminal symbols
- parent-children relations among nodes depend from the rules defined by the grammar

Grammar Definition

Context Free Grammar

A Context Free Grammar is a tuple $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ where:

- $\mathcal{V}_\mathcal{T}$ is a finite non-empty set of terminal symbols (alphabet)
- $\mathcal{V}_\mathcal{N}$ is a finite non-empty set of non-terminal symbols s.t. $\mathcal{V}_\mathcal{N} \cap \mathcal{V}_\mathcal{T} = \varnothing$
- $\mathcal{S}$ is the start symbol of the grammar s.t. $\mathcal{S} \in \mathcal{V}_\mathcal{N}$
- $\mathcal{P}$ is a finite non-empty set of productions s.t. $\mathcal{P} \subseteq \mathcal{V}_\mathcal{N} \times \mathcal{V}^*$ where $\mathcal{V}^* = \mathcal{V}_\mathcal{T} \cup \mathcal{V}_\mathcal{N}$

Parsing Example

Expressions grammar I

$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid id$

Find the sequence or productions for the string "$id + id * id$" and derive the corresponding parse tree

Expressions grammar II

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T/F \mid F$

$F \rightarrow (E) \mid id$

Parsing Example

Expressions grammar I

$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid id$

Find the sequence or productions for the string "$id + id * id$" and derive the corresponding parse tree

Expressions grammar II

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T/F \mid F$
$F \rightarrow (E) \mid id$

Derivations

Derivation

The construction of a parse tree can be made precise by taking a derivational view, in which production are considered as rewriting rules.

A sentence belongs to a language if there is a derivation from the initial symbol to the sentence.
e.g. $E \rightarrow E + E|E * E| - E|(E)|\textbf{id}$

Kind of derivations

Each sentence can be generated according to two different strategies leftmost and rightmost. Parsers generally return one of this two derivations.

Derivations

Derivation

The construction of a parse tree can be made precise by taking a derivational view, in which production are considered as rewriting rules.

A sentence belongs to a language if there is a derivation from the initial symbol to the sentence.
e.g. $E \rightarrow E + E | E * E | - E | (E) | \mathbf{id}$

Kind of derivations

Each sentence can be generated according to two different strategies leftmost and rightmost. Parsers generally return one of this two derivations.

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguos. An ambiguous grammar has more then one left-most derivation or more than one rightmost derivation for the same sentence.

Ambiguity and Precedence of Operators

Using the simplest grammar for expressions let's derive again the parse tree for:

$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$

Now consider the following grammar:

$E \rightarrow E + T | E - T | T$

$T \rightarrow T * F | T / F | F$

$F \rightarrow (E) | \mathbf{id}$

Use of ambiguos grammar

In some case it can be convenient to use ambiguous grammar, but then it is necessary to define precise disambiguating rules

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguos. An ambiguous grammar has more then one left-most derivation or more than one rightmost derivation for the same sentence.

Ambiguity and Precedence of Operators

Using the simplest grammar for expressions let's derive again the parse tree for:

$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$

Now consider the following grammar:
$E \rightarrow E + T | E - T | T$
$T \rightarrow T * F | T / F | F$
$F \rightarrow (E) | \mathbf{id}$

Use of ambiguos grammar

In some case it can be convenient to use ambiguous grammar, but then it is necessary to define precise disambiguating rules

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguos. An ambiguous grammar has more then one left-most derivation or more than one rightmost derivation for the same sentence.

Ambiguity and Precedence of Operators

Using the simplest grammar for expressions let's derive again the parse tree for:

$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$

Now consider the following grammar:

$E \rightarrow E + T|E - T|T$
$T \rightarrow T * F|T/F|F$
$F \rightarrow (E)|\mathbf{id}$

Use of ambiguos grammar

In some case it can be convenient to use ambiguous grammar, but then it is necessary to define precise disambiguating rules

Ambiguity

Conditional statements

Consider the following grammar:

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \textbf{other}
\end{array}
$$

decide if the following sentence belongs to the generated language:

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

Exercises

Consider the grammar:

$$S \to SS + |SS * |a$$

and the string $aa + a*$

- Give the leftmost derivation for the string
- Give the rightmost derivation for the string
- Give a parse tree for the string
- Is the grammar ambiguous or unambiguous?
- Describe the language generated by this grammar?

Define grammars for the following languages:

- $\mathscr{L} = \{w \in \{0,1\}^* | w \text{ is palindrom}\}$
- $\mathscr{L} = \{w \in \{0,1\}^* | w \text{ contains the same occurrences of 0 and 1}\}$

Exercises

Consider the grammar:

$$S \rightarrow SS + |SS * |a$$

and the string $aa + a*$

- Give the leftmost derivation for the string
- Give the rightmost derivation for the string
- Give a parse tree for the string
- Is the grammar ambiguous or unambiguous?
- Describe the language generated by this grammar?

Define grammars for the following languages:

- $\mathscr{L} = \{w \in \{0,1\}^* | w \text{ is palindrom}\}$
- $\mathscr{L} = \{w \in \{0,1\}^* | w \text{ contains the same occurrences of 0 and 1}\}$

Type of parsers

Three general type of parsers:

- universal (any kind of grammar)
- top-down
- bottom-up

Chomsky Hierarchy

A hierarchy of grammars can be defined imposing constraints on the structure of the productions in set $\mathcal{P}$ ($\alpha, \beta, \gamma \in \mathcal{V}^*$, $a \in \mathcal{V}_T$, $A, B \in \mathcal{V}_N$):

T0. Unrestricted Grammars:
- Production Schema: *no constraints*
- Recognizing Automaton: Turing Machines

T1. Context Sensitive Grammars:
- Production Schema: $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Recognizing Automaton: Linear Bound Automaton (LBA)

T2. **Context-Free Grammars**:
- Production Schema: $A \rightarrow \gamma$
- Recognizing Automaton: Non-deterministic Push-down Automaton

T3. Regular Grammars:
- Production Schema: $A \rightarrow a$ or $A \rightarrow aB$
- Recognizing Automaton: Finite State Automaton

Push-down Automata

Definition

A Push-down Automaton is a tuple $\langle \Sigma, \Gamma, \mathcal{Z}_0, \mathcal{S}, s_0, \mathcal{F}, \delta \rangle$ where:

- Σ defines the input alphabet
- Γ defines the alphabet for the stack
- $\mathcal{Z}_0 \in \Gamma$ is the symbol that is put initially on the stack
- $\mathcal{S}$ represents the set of states
- $s_0 \in \mathcal{S}$ is the initial state of the automaton
- $\mathcal{F} \subseteq \mathcal{S}$ is the set of final states
- $\delta : \mathcal{S} \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \ldots$ represents the transition function

Deterministic vs. Non-Deterministic

Push-down automata can be defined according to a deterministic strategy or a non-deterministic one. In the first case the transition function returns elements in the set $\mathcal{S} \times \Gamma^*$, in the second case the returned element belongs to the set $\mathscr{P}(\mathcal{S} \times \Gamma^*)$

Push-down Automata - How do they proceed?

Intuition

- The automaton starts with an empty stack and a string to read
- On the base of its status (state, symbol at the top of the stack), and of the character at the beginning of the input string it changes its status consuming the character from the input string.
- The status change consists in the insertion of one or more symbol in the stack after having removed the one at the top, and in the transition to another internal state
- the string is accepted when all the symbols in the input stream have been considered and the automaton reach a status in which the state is final or the stack is empty

Push-down Automata

Configuration

Given a Push-dow Automaton $\mathcal{A} = \langle \Sigma, \Gamma, \mathcal{Z}_0, \mathcal{S}, s_0, \mathcal{F}, \delta \rangle$ a configuration is given by the tuple $\langle s, x, \gamma \rangle$ where:

- $s \in \mathcal{S}, x \in \Sigma^*, \gamma \in \Gamma^*$

The configuration of an automaton represent its global state and contains the information to know its future states.

Transition

Given $\mathcal{A} = \langle \Sigma, \Gamma, \mathcal{Z}_0, \mathcal{S}, s_0, \mathcal{F}, \delta \rangle$ and two configurations $\chi = \langle s, x, \gamma \rangle$ and $\chi' = \langle s', x', \gamma' \rangle$ it can happen that the automaton passes from the first configuration to the second ($\chi \vdash_A \chi'$) iff:

- $\exists a \in \Sigma . x = ax'$
- $\exists Z \in \Gamma, \eta, \sigma \in \Gamma^* . \gamma = Z\eta \wedge \gamma' = \sigma\eta$
- $\delta(s, a, Z) = (s', \sigma)$

Push-down Automata

Acceptance by empty stack

Given $\mathcal{A} = \langle \Sigma, \Gamma, \mathcal{Z}_0, \mathcal{S}, s_0, \mathcal{F}, \delta \rangle$ a configuration $\chi = \langle s, x, \gamma \rangle$ accepts a string iff $x = \gamma = \epsilon$

Acceptance by final state

Given $\mathcal{A} = \langle \Sigma, \Gamma, \mathcal{Z}_0, \mathcal{S}, s_0, \mathcal{F}, \delta \rangle$ a configuration $\chi = \langle s, x, \gamma \rangle$ accepts a string iff $x = \epsilon$ and $s \in \mathcal{F}$

Push-down Automata - Exercise

- Define a push-down automaton that accept the language $\mathcal{L} = \{a^n b^n | n \in \mathbb{N}^+\}$
- Define a push-down automaton that accept the language $\mathcal{L} = \{w\overline{w} | w \in \{a, b\}^+\}$
- Define a push-down automaton that accept the language $\mathcal{L} = \{a^n b^m c^{2n} | n \in \mathbb{N}^+ \wedge m \in \mathbb{N}\}$

Push-down Automata - Exercise

- Define a push-down automaton that accept the language $\mathcal{L} = \{a^n b^n | n \in \mathbb{N}^+\}$
- Define a push-down automaton that accept the language $\mathcal{L} = \{w\overline{w} | w \in \{a, b\}^+\}$
- Define a push-down automaton that accept the language $\mathcal{L} = \{a^n b^m c^{2n} | n \in \mathbb{N}^+ \land m \in \mathbb{N}\}$

Push-down Automata - Exercise

- Define a push-down automaton that accept the language $\mathcal{L} = \{a^n b^n | n \in \mathbb{N}^+\}$
- Define a push-down automaton that accept the language $\mathcal{L} = \{w\overline{w} | w \in \{a, b\}^+\}$
- Define a push-down automaton that accept the language $\mathcal{L} = \{a^n b^m c^{2n} | n \in \mathbb{N}^+ \wedge m \in \mathbb{N}\}$

CF grammars are capable to describe the syntax of most, but not all, the programming languages. For instance, the requirement that identifiers must be declared before their usage cannot be expressed in CF grammars.

So what we can do?

Ambiguity

- Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars.
- A language that only admits ambiguous grammars is called an e.g. $\{a^n b^m c^k \mid n = m \text{ or } m = k, n, m, k \geq 0\}$
- A Turing machine cannot decide whether a context-free language is inherently ambiguous or not.

CF grammars are capable to describe the syntax of most, but not all, the programming languages. For instance, the requirement that identifiers must be declared before their usage cannot be expressed in CF grammars.

So what we can do?

Ambiguity

- Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars
- A language that only admits ambiguous grammars is called an inherently ambiguous language, e.g. $\{a^n\, b^m\, c^k \mid n = m \text{ or } m = k; n, m, k \geq 0\}$
- A Turing machine cannot decide whether a context-free language is inherently ambiguous or not

CF grammars are capable to describe the syntax of most, but not all, the programming languages. For instance, the requirement that identifiers must be declared before their usage cannot be expressed in CF grammars.

So what we can do?

Ambiguity

- Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars
- A language that only admits ambiguous grammars is called an inherently ambiguous language, e.g. $\{a^n b^m c^k \mid n = m \text{ or } m = k; n, m, k \geq 0\}$
- A Turing machine cannot decide whether a context-free language is inherently ambiguous or not

CF grammars are capable to describe the syntax of most, but not all, the programming languages. For instance, the requirement that identifiers must be declared before their usage cannot be expressed in CF grammars.

So what we can do?

Ambiguity

- Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars
- A language that only admits ambiguous grammars is called an inherently ambiguous language, e.g. $\{a^n\, b^m\, c^k \mid n = m \text{ or } m = k; n, m, k \geq 0\}$
- A Turing machine cannot decide whether a context-free language is inherently ambiguous or not

CF grammars are capable to describe the syntax of most, but not all, the programming languages. For instance, the requirement that identifiers must be declared before their usage cannot be expressed in CF grammars.

So what we can do?

Ambiguity

- Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars
- A language that only admits ambiguous grammars is called an inherently ambiguous language, e.g. $\{a^n\, b^m\, c^k \mid n = m \text{ or } m = k; n, m, k \geq 0\}$
- A Turing machine cannot decide whether a context-free language is inherently ambiguous or not

ToC

ToC

Left Recursion

Left recursive grammars

A grammar $\mathscr{G}$ is left recursive if it has a non terminal A such that there is a derivation $A \stackrel{*}{\Longrightarrow} A\alpha$ for some sting α. Top-down parsing strategies cannot handle left-recursive grammars

Immediate left recursion

A grammar as an immediate left recursion if there is at least one production of the form $A \rightarrow A\alpha$. It is possible to transform the grammar still generating the same language and removing the left recursion. Consider the generale case:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where $n, m \geq 1$ and all β_i do not start with A. Equivalent productions are:

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

Left Recursion

Left recursive grammars

A grammar $\mathscr{G}$ is left recursive if it has a non terminal A such that there is a derivation $A \stackrel{*}{\Longrightarrow} A\alpha$ for some sting α. Top-down parsing strategies cannot handle left-recursive grammars

Immediate left recursion

A grammar as an immediate left recursion if there is at least one production of the form $A \rightarrow A\alpha$. It is possible to transform the grammar still generating the same language and removing the left recursion. Consider the generale case:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where $n, m \geq 1$ and all β_i do not start with A. Equivalent productions are:

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

Eliminating Left Recursion

The following is a general algorithm to eliminate left recursion at any level

Input: Grammar G with no cycles or $\epsilon - productions$
Output: An equivalent grammar with no left recursion
Arrange the non terminals in some order $A_1, A_2, ..., A_n$
for all $i \in [1...n]$ **do**
  for all $j \in [1...i-1]$ **do**
    replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$ where $A_j \rightarrow \delta_1|\delta_2|\cdots|\delta_k$ are all current $A_j - productions$
  end for
  eliminate the immediate left recursion among the $A_i - productions$
end for

Left Factoring

Left Factoring

Left Factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice

Transformation rule

In general the grammar:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

can be rewritten in:

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \\ A' & \rightarrow & \beta_1 | \beta_2 \end{array}$$

In general find the longest prefix and then iterate till no two alternatives for a nonterminal have a common prefix

Left Factoring

Left Factoring

Left Factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice

Transformation rule

In general the grammar:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

can be rewritten in:

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \\ A' & \rightarrow & \beta_1 | \beta_2 \end{array}$$

In general find the longest prefix and then iterate till no two alternatives for a nonterminal have a common prefix

Top-down parsing

Top-down parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string starting from the root and creating the nodes of the parse tree in pre-order (depth-first). Equivalently . . . finding the left-most derivation for an input string.

Recursive descent parsing

A recursive descent (top-down) parsing consist of a set of procedures, one for each nonterminal.

function A
  Choose an *A-production*, $A \rightarrow X_1 X_2 \cdots X_k$;
  for all $i \in [1 \cdots k]$ **do**
    if (X_i is a non terminal) **then** call procedure $X_i()$;
    else if (X_i equals the current input symbol a) **then**
      advance the input to the next symbol;
    else an error has occurred;
    end if
  end for
end function

Top-down parsing

Backtracking is expensive and not easy to manage. With grammar with no left-factoring and left-recursion we can do better:

At work

At each step of a top-down parsing the key problem is that of determining the production to be applied for a nonterminal.
Let's consider the usual sentence **id** $+$ **id** $*$ **id** and a suitable grammar for top-down parsing:
$E \rightarrow TE' \quad E' \rightarrow +TE'|\epsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT'|\epsilon \quad F \rightarrow (E)|\mathbf{id}$

FIRST and FOLLOW sets

$FIRST(\alpha)$	set of terminals that begin strings derived from α
$FOLLOW(A)$	set of terminals a that can appear immediately to the right of A in some sentential form
$nullable(X)$	it is true if it is possible to derive ϵ from X

FIRST

To compute $FIRST(X)$ for all grammar symbols X, apply the following rules until no more terminals or ϵ can be addedd to any $FIRST$ set

1. if X is a terminal, then $FIRST(X) = \{X\}$
2. if X is a non terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $FIRST(X)$ if a is in $FIRST(Y_i)$, for some $i \leq k$, and ϵ is in all of $FIRST(Y_1) \cdots FIRST(Y_{i-1})$. If ϵ is in $FIRST(Y_j)$ for all $j = 1, 2, \ldots, k$ then add ϵ to $FIRST(X)$. If Y_1 does not derive ϵ, then we add nothing more to $FIRST(X)$, but if $Y_1 \rightarrow^* \epsilon$, then we add $FIRST(Y_2)$, and so on.
3. if $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$

It is then possible to compute $FIRST$ for any string $X_1 X_2 \cdots X_k$

FIRST and FOLLOW sets

$FIRST(\alpha)$	set of terminals that begin strings derived from α
$FOLLOW(A)$	set of terminals a that can appear immediately to the right of A in some sentential form
$nullable(X)$	it is true if it is possible to derive ϵ from X

FIRST

To compute $FIRST(X)$ for all grammar symbols X, apply the following rules until no more terminals or ϵ can be addedd to any *FIRST* set

1. if X is a terminal, then $FIRST(X) = \{X\}$
2. if X is a non terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $FIRST(X)$ if a is in $FIRST(Y_i)$, for some $i \leq k$, and ϵ is in all of $FIRST(Y_1) \cdots FIRST(Y_{i-1})$. If ϵ is in $FIRST(Y_j)$ for all $j = 1, 2, \ldots, k$ then add ϵ to $FIRST(X)$. If Y_1 does not derive ϵ, then we add nothing more to $FIRST(X)$, but if $Y_1 \rightarrow^* \epsilon$, then we add $FIRST(Y_2)$, and so on.
3. if $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$

It is then possible to compute *FIRST* for any string $X_1 X_2 \cdots X_k$

FIRST and FOLLOW sets

FOLLOW

To compute $FOLLOW(A)$ for all non terminals A, apply the following rules until nothing can be added to any *FOLLOW* set

1. Place \$ in $FOLLOW(S)$, where S is the start symbol, and \$ is the input right endmarker.
2. if there is a production $A \to \alpha B \beta$, then everything in $FIRST(\beta)$ except ϵ is in $FOLLOW(B)$
3. if there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is in $FOLLOW(B)$

FIRST and FOLLOW sets

Derive *FIRST*, *FOLLOW*, *nullable* sets for the expression grammar
Now consider the following grammar:

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

Parsing table

The parsing table is a two dimension array in which rows a nonterminal symbols and columns are terminal symbols plus \$. In each cell a production is then stored (determinism).

Construction of the Parsing Table

Input: Grammar $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$
Output: Parsing table M
for all $A \to \alpha \in \mathcal{P}$ **do**
  for all $a \in \mathit{FIRST}(\alpha) \backslash \{\epsilon\}$ **do**
    add $A \to \alpha$ to $M[A, a]$
  end for
  if $\epsilon \in \mathit{FIRST}(\alpha)$ **then**
    for all $b \in \mathit{FOLLOW}(A)$ **do** `// b can be $`
      add $A \to \alpha$ to $M[A, b]$
    end for
  end if
end for

FIRST and FOLLOW sets

Derive the parsing table for the expresion grammar:

$$E \rightarrow TE' \quad E' \rightarrow +TE'|\epsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT'|\epsilon \quad F \rightarrow (E)|id$$

	FIRST	**FOLLOW**	**Null.**
E	$(, id$	$), \$$	
E'	$+$	$),\$$	yes
T	$(, id$	$+,), \$$	
T'	$*$	$+,), \$$	yes
F	$(, id$	$*, +,), \$$	

LL(1) Grammars

LL(k)

Predictive parsing that does not need backtracking. The first **L** stands for Left-to-right and the second **L** stands for Leftmost and **k** indicates the maximum number of lookahead symbols needed to take a decision

Most programming constructs can be expressed using an $LL(1)$ grammar. A grammar G is $LL(1)$ iff whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. for no terminal a do both α and β derive strings beginning with a
2. at most one of α and β can derive the empty string
3. if $\beta \to^* \epsilon$, then α does not derive any string starting with a terminal in $FOLLOW(A)$. Likewise if $\alpha \to^* \epsilon$, then β does not derive any string starting with a terminal in $FOLLOW(A)$

LL(1) Grammars

LL(k)

Predictive parsing that does not need backtracking. The first **L** stands for Left-to-right and the second **L** stands for Leftmost and **k** indicates the maximum number of lookahead symbols needed to take a decision

Most programming constructs can be expressed using an $LL(1)$ grammar. A grammar G is $LL(1)$ iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. for no terminal a do both α and β derive strings beginning with a
2. at most one of α and β can derive the empty string
3. if $\beta \rightarrow^* \epsilon$, then α does not derive any string starting with a terminal in $FOLLOW(A)$. Likewise if $\alpha \rightarrow^* \epsilon$, then β does not derive any string starting with a terminal in $FOLLOW(A)$

Parsing table

Derive *FIRST*, *FOLLOW*, *nullable* sets and parsing table for the following grammar:

$$S \to iEtSS'|a \quad S' \to eS|\epsilon \quad E \to b$$

Parsing table:

	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Parsing table

Derive *FIRST*, *FOLLOW*, *nullable* sets and parsing table for the following grammar:

$$S \to iEtSS' | a \quad S' \to eS | \epsilon \quad E \to b$$

Parsing table:

	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Non-recursive predictive parsing

Table-driven predictive parsing

Input: A string w and a parsing table M for a grammar $\mathcal{G}$
Output: if w is in $\mathscr{L}(\mathcal{G})$, a leftmost derivation of w, otherwise an error indication
init stack with \$ and add \$ at the end of w
push on top of the stack the start symbol of $\mathcal{G}$;
set *ip* to point to the first symbol of w;
set X to the current top stack symbol;
while ($X \neq \$$) **do**
  if (X is $w[ip]$) **then** pop the stack and advance *ip*;
  else if (X is a terminal) **then** error();
  else if ($M[X, w[ip]]$ is an error entry) **then** error();
  else if ($M[X, w[ip]] = X \to Y_1 Y_2 \cdots Y_k$) **then**
    output the production $X \to Y_1 Y_2 \cdots Y_k$;
    pop the stack;
    push $Y_k Y_{k-1} \cdots Y_1$ onto the stack, with Y_1 on top;
  end if
  Set X to the current top stack symbol;
end while

LL(1) parser moves (1/2)

MATCHED	STACK	INPUT	ACTION
	E$	**id** $+$ **id** $*$ **id**$	

LL(1) parser moves (2/2)

MATCHED	STACK	INPUT	ACTION
	S$	ibtibtaea$	

Error Recovery in Predictive Parsing

Error detection

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A,a]$ is ERROR.

Error Recovery in Predictive Parsing

Error detection

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A,a]$ is ERROR.

Panic Mode

Based on the idea of skipping symbols on the input until a token in a synchronizing set appears. Strategies:

- place all symbols in $FOLLOW(A)$ into the synchronizing set for nonterminal A.
- symbols starting higher level constructs
- use of ϵ-productions to change the symbol in the stack
- just pop the symbol in the stack and send alert

Error Recovery in Predictive Parsing

Error detection

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A,a]$ is ERROR.

Phrase-level recovery

Fill the blank entries in the predictive parsing table with entries to recovery routines.

ToC

Bottom-up Parsing

Bottom-up Parsing

The problem of Bottom-up parsing can be viewed as the problem of constructing a parse tree for an input string beginning at the leaves and working up towards the root.
In particular we will consider the problem of finding the rightmost derivation given an input string, through a series of reductions to reach the initial symbol

Let's consider the input string **id** $*$ **id** using the simple grammar for expressions

Bottom-up Parsing

Bottom-up Parsing

The problem of Bottom-up parsing can be viewed as the problem of constructing a parse tree for an input string beginning at the leaves and working up towards the root.
In particular we will consider the problem of finding the rightmost derivation given an input string, through a series of reductions to reach the initial symbol

Let's consider the input string **id** $*$ **id** using the simple grammar for expressions

Tools for Bottom-up Parsing

Reductions

In a bottom-up parser at each step a reduction is applied. A certain string is reduced to the head of the production (non-terminal) applying the production in reverse. The key decision is when to reduce!

Handle Pruning

A handle is a substring of a sentential form that matches the body of a production and whose reduction represents a step along the rightmost derivation of the sentential form in reverse.

Consider the grammar $S \to 0S1|01$ and the two sentential forms 000111, $00S11$

Tools for Bottom-up Parsing

Reductions

In a bottom-up parser at each step a reduction is applied. A certain string is reduced to the head of the production (non-terminal) applying the production in reverse. The key decision is when to reduce!

Handle Pruning

A handle is a substring of a sentential form that matches the body of a production and whose reduction represents a step along the rightmost derivation of the sentential form in reverse.

Consider the grammar $S \rightarrow 0S1|01$ and the two sentential forms 000111, $00S11$

Shift-reduce parsing

Shift-reduce parsing

A shift-reduce parser is a particular kind of bottom-up parser in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. Four possible actions are possible:

- shift
- reduce
- accept
- error

Conflicts

- shift/reduce
- reduce/reduce

Shift-reduce parsing

Shift-reduce parsing

A shift-reduce parser is a particular kind of bottom-up parser in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. Four possible actions are possible:

- shift
- reduce
- accept
- error

Conflicts

- shift/reduce
- reduce/reduce

Shift-reduce parsing

Consider the grammar $S \rightarrow SS + | SS * | a$ and the following sentential forms:
$SSS + a * +$, $SS + a * a+$, $aaa * a + +$

LR Parsing

LR Parsers

LR parsers show interesting good properties:

- all programming languages admit a grammar that can be parsed by an LR parser
- most general non-backtracking shift-reduce parser
- syntactic errors can be detected as soon as it is possible to do so on a left-to right scan of the input
- the class of grammars that can be parsed by an LR is a proper superset of that parsable with a predictive parsing strategy

Items and LR(0) Automaton

Item

An Item is a production in which a dot $\cdot$ has been added in the body. Intuitively, it indicates how much of a production we have seen during parsing.

One collection of sets of $LR(0)$ items, called the canonical $LR(0)$ collection, provides the basis for constructing a DFA that is used to make decisions.

The construction of the canonical $LR(0)$ is based on two functions $\texttt{CLOSURE}$ and $\texttt{GOTO}$

CLOSURE

If $\mathcal{I}$ is a set of items for a grammqr $\mathcal{G}$, then $\mathrm{CLOSURE}(\mathcal{I})$ is the set of items constructed from $\mathcal{I}$ by the following two rules:

1. Initially, add every item in $\mathcal{I}$ to $\mathrm{CLOSURE}(\mathcal{I})$
2. if $A \to \alpha \cdot B\beta$ is in $\mathrm{CLOSURE}(\mathcal{I})$ and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to $\mathrm{CLOSURE}(\mathcal{I})$, if is not already there. Apply this rule until no more items can be added to $\mathrm{CLOSURE}(\mathcal{I})$

CLOSURE

Consider the expression grammar:
$E' \to E \quad E \to E + T|T \quad T \to T * F|F \quad F \to (E)|id$
Compute the closure of the item $E' \to \cdot E$

GOTO

GOTO(I,X)

GOTO(I,X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I.

- Intuitively the GOTO function is used to define the transition of the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I,X) specifies the transition from the state for I under input X

Build the LR(0) automaton

Build the LR(0)automaton for the expression grammar:
$E' \to E \quad E \to E + T|T \quad T \to T * F|F \quad F \to (E)|id$

Use of the LR(0) automaton

The LR(0) automaton can be used for deriving a parsing table, which has a number of states equal to the states of the LR(0) automaton and the actions are dependent from the action of the automaton itself. The parsing table will have two different sections, one named ACTION and the other GOTO:

Parsing table

1. The ACTION table has a row for each state of the LR(0) automaton and a column for each terminal symbol. The value of ACTION[i,a] can have one of for forms:
   1. Shift j where j is a state (generally abbreviated as Sj).
   2. Reduce $A \rightarrow \beta$. The action of the parser reduces β to A in the stack (generally abbreviated as $R(A \rightarrow \beta)$)
   3. Accept
   4. Error
2. The GOTO table has a row for each state of the LR(0) automaton and a column for each nonterminal. The value of GOTO[I_i,A] = I_j if the GOTO function maps set of items accordingly on the LR(0) automaton

LR(0) table construction

LR(0) table

The LR(0) table is built according to the following rules, where "*i*" is the considered state and "*a*" a symbol in the input alphabet:

1. $\texttt{ACTION}[i,a] \leftarrow$ shift j
   if $[A \to \alpha \cdot a\beta]$ is in state i and $\texttt{GOTO}(i,a) = j$ – (generally represented as Sj)
2. $\texttt{ACTION}[i,*] \leftarrow$ reduce($A \to \beta$)
   if state i includes the item $(A \to \beta\cdot)$ – (generally represented as R($A \to \beta$))
3. $\texttt{ACTION}[i,*] \leftarrow$ accept
   if the state includes the item $S' \to S\cdot$
4. $\texttt{ACTION}[i,*] \leftarrow$ error
   in all the other situations

LR(0) table construction

Consider the following grammars and sentences:
$S \rightarrow CC \quad C \rightarrow cC|d$ sentence: “ccd” and “ddd”

LR(0) table construction

Consider the following grammars and sentences:
$S \rightarrow aS|Ba \quad B \rightarrow Ba|b$ sentence: “aaba”

Use of the LR(0) automaton

Consider the string **id*id** and parse it

STACK	SYMBOLS	INPUT	ACTION
0	$	**id*id**$	$\cdots$

LR Parsing algorithm

General LR parsing program

The initial state of the parser is s_0 for the state and w (the whole string) on the input buffer.

Let a be the first symbol of w\$;
while true **do**
  let s be the state on top of the stack;
  if (ACTION[s,a] $=$ shift t) **then**
    push t onto the stack;
    let a be the next input symbol;
  else if (ACTION[s,a] $=$ reduce $A \to \beta$) **then**
    pop $|\beta|$ off the stack;
    let state t now be on top of the stack;
    push GOTO[t,A] onto the stack;
    output the production $A \to \beta$;
  else if (ACTION[s,a] $=$ accept) **then** break;
  else call error-recovery routine;
  end if
end while

SLR table construction

SLR(1) table

The LR(0) table is built according to the following rules, where "*i*" is the considered state and "*a*" a symbol in the input alphabet:

1. $\texttt{ACTION}[i,a] \leftarrow$ shift j
   if $[A \rightarrow \alpha \cdot a\beta]$ is in state i and $\texttt{GOTO}(i,a) = j$
2. $\texttt{ACTION}[i,a] \leftarrow$ reduce$(A \rightarrow \beta)$
   forall a in $\texttt{FOLLOW}(A)$ and if state i includes the item $(A \rightarrow \beta\cdot)$
3. $\texttt{ACTION}[i,\$] \leftarrow$ accept
   if the state includes the item $S' \rightarrow S\cdot$
4. $\texttt{ACTION}[i,*] \leftarrow$ error
   in all the other situations

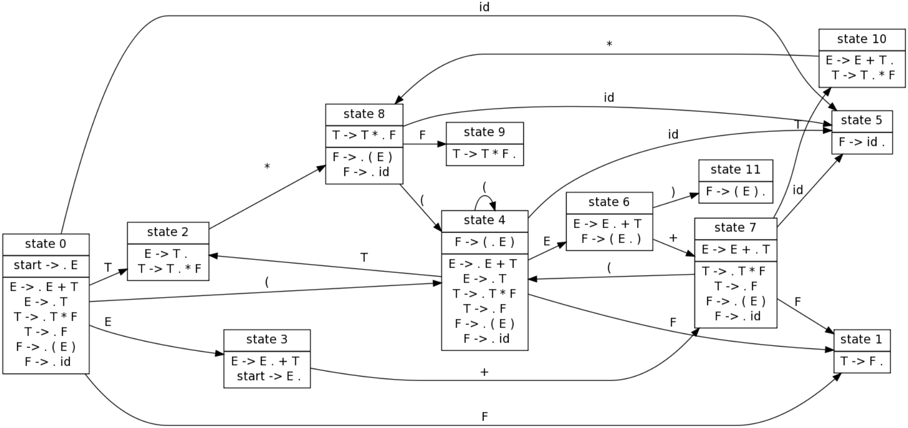
SLR table construction

Consider the following grammars and sentences:
$S \rightarrow aS|Ba \;\; B \rightarrow Ba|b$ sentence: “aaba”

LR(0) vs. SLR parsing

Consider the usual expression grammar:
$E' \to E \quad E \to E + T|T \quad T \to T * F|F \quad F \to (E)|id$
build LR(0) and SLR tables for the grammar, and then parse the sentence:

id$*$id+id



http://smlweb.cpsc.ucalgary.ca/start.html

LL(1) vs. SLR(1)

Consider the following grammars:

- $S \rightarrow AaAb | BbBa \quad A \rightarrow \epsilon \quad B \rightarrow \epsilon$
- $S \rightarrow SA | A \quad A \rightarrow a$

Build parsing tables for LL(1) and SLR(1)

Towards more powerful parsers

Consider the following grammar and derive the SLR parsing table:
$S \rightarrow L = R | R \quad L \rightarrow *R | id \quad R \rightarrow L$

Towards more powerful parsers

Viable prefix

A Viable prefix is a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.
We say item $A \to \beta_1 \cdot \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$.

LR parsers with lookahead

In order to enlarge the class of grammars that can be parsed we need to consider more powerful parsing strategies. In particular we will study:

- LR(1) parsers
- LALR parsers

LR(1) items

LR(1) items structure

The very general idea is to encapsulate more information in the items of an automaton to decide when to reduce. The solution is to differentiate items on the base of lookaheads. As a result a general item follows now the template $[A \rightarrow \alpha \cdot \beta, a]$

LR(1) items and reductions

Given the new form on an item, the parser will call for a reduction $A \rightarrow \alpha$ only for item sets including the item $[A \rightarrow \alpha \cdot, a]$ and only for symbol a

LR(1) CLOSURE and GOTO functions

Closure of an item

If $[A \to \alpha \cdot B\beta, a]$ is in I then for each production $B \to \gamma$ and for each terminal b in FIRST(βa) add the item $[B \to \cdot\gamma, b]$

GOTO(I, X)

Let J initially empty. For each item $[A \to \alpha \cdot X\beta, a]$ in I add item $[A \to \alpha X \cdot \beta, a]$ to set J. Then compute CLOSURE(J)

Consider the starting item as the closure of the item $[S' \to S, \$]$.

Exercise

Compute the LR(1) item sets for the following grammar:
$S \to CC \quad C \to cC|d$

LR(1) parsing table

How to build the LR(1) parsing table

1. build the collection of sets of LR(1) items for the grammar
2. Parsing actions for state i are:
   1. if $[A \rightarrow \alpha \cdot a\beta, b]$ is in I_i and GOTO(I_i, a)$=$ I_j then set ACTION[i, a] to shift J.
   2. if $[A \rightarrow \alpha\cdot, a]$ is in I_i $A \neq S'$ then set ACTION[i, a] to reduce($A \rightarrow \alpha$)
   3. if $[S' \rightarrow S\cdot, \$]$ is in I_i then set ACTION[$i, \$$] to accept
3. if $\texttt{GOTO}(I_i, A) = I_j$ then $\texttt{GOTO}[i, A] = j$
4. All entries not defined so far are mare "error"
5. The initial state of the parse is the one constructed from the set of items containing $[S' \rightarrow \cdot S, \$]$

Consider the following grammar and derive the LR(1) parsing table:
$S \rightarrow L = R | R \quad L \rightarrow *R | id \quad R \rightarrow L$

LR(1) parsing

Consider the following grammar and discuss applicability of LR(1) parsing:
$S \rightarrow aSa \mid a$

- Which is the language generated?
- Propose an alternative grammar parsable using an LR(1) parser

LR(1) parsing

Consider the following grammar and discuss applicability of LR(1) parsing:
$S \rightarrow aSa \mid a$

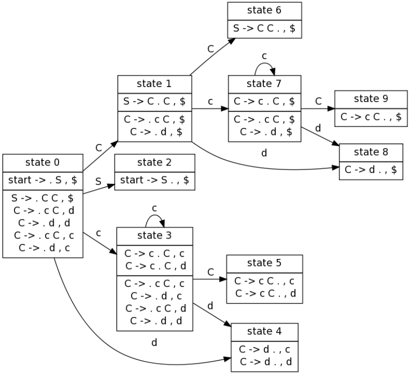
- Which is the language generated?
- Propose an alternative grammar parsable using an LR(1) parser

LALR parsing

- LR(1) for a real language a SLR parser has several hundred states. For the same language an LR(1) parser has several thousand states
- Can we produce a parser with power similar to LR(1) and table dimension similar to SLR?

LALR parsing

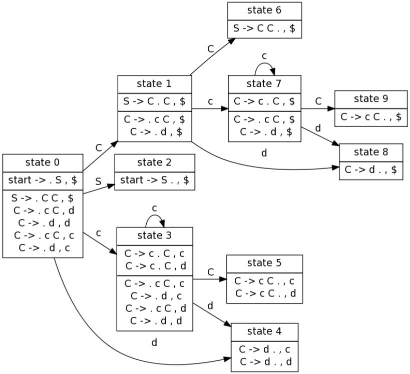
Let's consider the LR(1) automaton for the grammar
$S \rightarrow CC \quad C \rightarrow cC|d$



LALR table can be built from LR(1) automaton merging "similar" item sets.

LALR parsing

Let's consider the LR(1) automaton for the grammar
$S \rightarrow CC \quad C \rightarrow cC|d$



LALR table can be built from LR(1) automaton merging "similar" item sets.

Exercises

Consider the grammar:
$S \rightarrow Aa|bAc|dc|bda \quad A \rightarrow d$
show that is LALR(1) but not SLR(1)

Consider the grammar:
$S \rightarrow Aa|bAc|Bc|bBa \quad A \rightarrow d \quad B \rightarrow d$
show that is LR(1) but not LALR(1)

Exercises

Consider the grammar:
$S \rightarrow Aa|bAc|dc|bda \quad A \rightarrow d$
show that is LALR(1) but not SLR(1)

Consider the grammar:
$S \rightarrow Aa|bAc|Bc|bBa \quad A \rightarrow d \quad B \rightarrow d$
show that is LR(1) but not LALR(1)