1 Determinism and Parsing

The *parsing problem* is, given a string w and a context-free grammar G, to decide if $w \in L(G)$, and if so, to construct a derivation or a parse tree for w.

Parsing is studied in courses in compilers. To be efficient on large programs, parsing has to be linear time or nearly linear time. Parsing is often based on deterministic push-down automata.

1.1 Deterministic push-down automata

Definition 1.1 *A push-down automaton is* deterministic *if for each configuration at most one transition can apply.*

- This differs from deterministic finite automata because it is possible for no transition to apply so that the push-down automaton gets stuck in the middle of the input.
- It is not immediately obvious how to determine if a push-down automaton is deterministic. Here are some ways in which determinism can fail:

$((p, a, \epsilon), (q, \gamma))$ $((p, \epsilon, A), (q', \gamma'))$	Both transitions apply if the state is p, reading an a, and A is on top of the stack
$((p, a, A), (q, \gamma))$ $((p, a, AB), (q', \gamma'))$	Both transitions apply if the state is p, reading an a, and AB is on top of the stack
$((p, a, A), (q, \gamma))$ $((p, \epsilon, AB), (q', \gamma'))$	Both transitions apply if the state is p, reading an a, and AB is on top of the stack

et cetera

For a push-down automaton to be deterministic, there has to be a *conflict* between every pair of distinct transitions. The conflict can either be

1. the transitions $((p_i, a_i, \beta_i), (q_i, \gamma_i))$ have different states p_i
2. the transitions both read different symbol a_i, neither of which is ϵ, or
3. the transitions have different β_i, neither of which is a prefix of the other.

1.2 Deterministic context-free languages

Definition 1.2 *A language $L \subseteq \Sigma^*$ is a* deterministic context-free language *if $L\$ = L(M)$ for some deterministic push-down automaton M, where \$ is a new symbol not in Σ. Here $L\$ = \{w\$: w \in L\}$.*

- The \$ permits the push-down automaton to detect the end of the string. This is realistic, and also can help in some cases.
- For example, $a^* \cup \{a^n b^n : n \geq 1\}$ is a deterministic context-free languge, and the end marker is needed so that it is not necessary to guess the end of the string.
- The initial sequence of a's has to be put on the stack in case a sequence of b's follows, and when the \$ is seen, then these a's on the stack can be popped. Without the end marker, it is necessary to guess the end of the string, introducing nondeterminism.

Not all context-free languages are deterministic.

- Later we will show that $\{a^n b^m c^p : m, n, p \geq 0 \text{ and } m \neq n \text{ or } m \neq p\}$ is not a deterministic context-free language.
- Intuitively, the push-down automaton has to guess at the beginning whether to compare a and b or b and c.

It turns out that any deterministic context-free language can be parsed in linear time, though this is not easy to prove, because a deterministic push-down automaton can still spend a lot of time pushing and popping the stack.

Theorem 1.1 *The class of deterministic context-free languages is closed under complement. Thus if $L \subseteq \Sigma^*$ is a deterministic context-free language, so is $\Sigma^* - L$.*

The idea of the proof is this:

- Suppose L is a deterministic context-free language. Then there is a deterministic push-down automaton M such that $L(M) = L$.
- The problem is that M may have some "dead" configurations from which there is no transition that applies. These have to be removed.

- So we modify M so that it has no dead configurations by adding transitions to it.
- We also have to remove looping configurations; it is possible that M may get stuck in an infinite loop in the middle of reading its input. Such infinite loops have to be removed.
- These changes ensure that M always reads to the end of its input string.
- Then basically one exchanges accepting and non-accepting states of the modified M, to get a push-down automaton recognizing the complement of M.

Corollary 1.1 *The context-free language* $L = \{a^n b^m c^p : m \neq n \text{ or } m \neq p\}$ *is not deterministic.*

Proof: Let $\overline{L}$ be the complement of L.

- If L were deterministic then $\overline{L}$ would also be deterministic context-free.
- Consider $L_1 = \overline{L} \cap \mathcal{L}(a^*b^*c^*)$.
- If $\overline{L}$ were deterministic context-free then L_1 would at least be context-free.
- But $L_1 = \{a^n b^n c^n : n \geq 0\}$ which is not even context-free.
- Therefore L is not deterministic context-free.

Corollary 1.2 *The deterministic context-free languages are a* proper subset *of all context-free languages.*

1.3 Parsing in Practice

- Knuth in 1965 developed the LR (left-to-right) parsers that can recognize any deterministic context-free language in linear time, using look-ahead. These parsers create rightmost derivations, bottom-up, but have large memory requirements.

- DeRemer in 1969 developed the $LALR$ parsers, which are simple LR parsers. These require less memory than the LR parsers, but are weaker.
- It is difficult to find correct, efficient $LALR$ parsers. They are used for some computer languages including Java, but need some hand-written code to extend their power.
- $LALR$ parsers are automatically generated by compiler compilers such as Yacc and GNU Bison. The C and C++ parsers of Gcc started as $LALR$ parsers, but were later changed to recursive descent parsers that construct leftmost derivations top-down.

1.4 Top-Down versus Bottom-up Parsing

Top down parsers begin at the start symbol and construct a derivation forwards to attempt to derive the given string. Bottom up parsers start at the string and attempt to construct a derivation backwards to the start symbol. First we will discuss top-down parsers.

1.5 Top-Down Parsing

The basic idea of top-down parsing is to use the construction of lemma 3.4.1 in the text to create a push-down automaton from a grammar, and then make the push-down automaton deterministic. There are several heuristics to make the push-down automaton deterministic:

1. Look-ahead one symbol.
2. Left factoring
3. Left recursion removal

Grammars for which one-symbol look-ahead suffices for top-down left-to-right parsing are called $LL(1)$ grammars. Let's recall lemma 3.4.1.

Lemma 1.1 (3.4.1) *The class of languages recognized by push-down automata is the same as the class of context-free languages.*

Proof: Given a context-free grammar $G = (V, \Sigma, R, S)$, one can construct a push-down automaton M such that $L(M) = L(G)$ as follows:

$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$ where Δ has the rules

(1)	$((p, \epsilon, \epsilon), (q, S))$	
(2)	$((q, \epsilon, A), (q, x))$	if $A \to x$ is in R (do leftmost derivation on the stack)
(3)	$((q, a, a), (q, \epsilon))$	for each $a \in \Sigma$ (remove matched symbols)

- The push-down automaton from lemma 3.4.1 constructs a left-most derivation.
- The idea of top-down parsing is to try to make this push-down automaton deterministic, both by modifying the grammar (left factoring and left recursion removal) and by modifying the push-down automaton (one-symbol look-ahead).
- We give three heuristics which may help to make the push-down automaton deterministic, but they do not always work.

1.6 Left Factoring

If in G we have productions of this form:

$$\begin{array}{ll} A \to \alpha\beta_1 & \alpha \neq \epsilon \\ A \to \alpha\beta_2 & n \geq 2 \\ \ldots & \\ A \to \alpha\beta_n & \end{array}$$

then these productions can be replaced by the following:

$$\begin{array}{l} A \to \alpha A' \\ A' \to \beta_1 \\ A' \to \beta_2 \\ \ldots \\ A' \to \beta_n \end{array}$$

where A' is a new nonterminal.

Example:

Replace $\begin{array}{l} A \to Ba \\ A \to Bb \end{array}$ by $\begin{array}{l} A \to BA' \\ A' \to a \\ A' \to b. \end{array}$ The idea is to delay the choice of which production to apply until a one-symbol lookahead can help to make the decision.

1.7 Left Recursion Removal

Suppose we have the rules

$$\begin{array}{ll} A \to A\alpha_1 & A \to \beta_1 \\ A \to A\alpha_2 & A \to \beta_2 \\ \ldots & \ldots \\ A \to A\alpha_n & A \to \beta_m. \end{array}$$

Then these rules can be replaced by the following:

$$\begin{array}{lll} A \to \beta_1 A' & A' \to \alpha_1 A' & A' \to \epsilon \\ A \to \beta_2 A' & A' \to \alpha_2 A' & \\ \ldots & \ldots & \\ A \to \beta_m A' & A' \to \alpha_n A'. & \end{array}$$

The problem is to know when to terminate the recursion. The idea is to change the structure of the recursion so that one can tell by look-ahead when to stop and how to recurse.

Example:

Replace $\begin{array}{ll} A \to Aa & A \to c \\ A \to Ab & A \to d, \end{array}$ by $\begin{array}{lll} A \to cA' & A' \to aA' & A' \to \epsilon. \\ A \to dA' & A' \to bA' & \end{array}$ Both sets of productions generate $(c \cup d)(a \cup b)^*$, but the second set generates the same strings in a different way.

1.8 One-symbol Look-ahead

The idea is to use the next symbol to decide which production to use. In the push-down automaton of lemma 3.4.1, even after applying the previous two heuristics, it may be difficult to decide among the following transitions:

$$((q, \epsilon, A), (q, x)) \text{ for } A \to x \text{ in } R.$$

For example, the grammar may have productions $S \to aSa$ and $S \to bSb$, so how does one decide between the productions $((q, \epsilon, S), (q, aSa))$ and $((q, \epsilon, S), (q, bSb))$? To handle this, new productions

$$((q, a, \epsilon), (q_a, \epsilon))$$

are added to the push-down automaton, for each $a \in \Sigma$.

The following transitions are also added:

$((q_a, \epsilon, a), (q, \epsilon))$	all $a \in \Sigma$ (analogous to type (3) transitions for the pda of lemma 3.4.1)
$((q_a, \epsilon, A), (q_a, x))$	if $A \to x$ is in R and $x \Rightarrow^* w$ for some w beginning with a or $x \Rightarrow^* \epsilon$ (analogous to type (2) transitions for the pda of lemma 3.4.1)

1.9 An example: One-symbol look-ahed

Let's look at a particular context-free language and see how the push-down automata with and without look-ahead differ for it. Consider this grammar:

$$G = (V, \Sigma, R, S)$$

$$\Sigma = \{a, b, c\}, V = \{S, a, b, c\}$$

$$R: \begin{array}{l} S \to aSa \\ S \to bSb \\ S \to c \end{array}$$

Push-down automaton without look-ahead:

$$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$$

Δ:

$((p, \epsilon, \epsilon), (q, S))$	
$((q, \epsilon, S), (q, aSa))$	These rules all
$((q, \epsilon, S), (q, bSb))$	apply at the same
$((q, \epsilon, S), (q, c))$	time, causing nondeterminism.
$((q, a, a), (q, \epsilon))$	
$((q, b, b), (q, \epsilon))$	
$((q, c, c), (q, \epsilon))$.	

Push-down automaton with look-ahead:

$$M = (\{p, q, q_a, q_b, q_c, q_\$\}, \Sigma, V, \Delta, p, \{q_\$\})$$

Δ: $((p, \epsilon, \epsilon), (q, S))$
$((q, a, \epsilon), (q_a, \epsilon))$
$((q, b, \epsilon), (q_b, \epsilon))$
$((q, c, \epsilon), (q_c, \epsilon))$
$((q, \$, \epsilon), (q_\$, \epsilon))$
$((q_a, \epsilon, a), (q, \epsilon))$
$((q_b, \epsilon, b), (q, \epsilon))$
$((q_c, \epsilon, c), (q, \epsilon))$
$((q_a, \epsilon, S), (q_a, aSa))$ These rules no longer
$((q_b, \epsilon, S), (q_b, bSb))$ apply at the same
$((q_c, \epsilon, S), (q_c, c))$ time, eliminating nondeterminism.

- It is easy to verify that this push-down automaton is deterministic by looking at the transitions.
- In this case there was no left factoring or left recursion removal done before adding one-symbol look-ahead to the push-down automaton.
- Now let's consider how the two push-down automata would behave on the input *abcba* and how the second one eliminates all nondeterminism.

For the push-down automaton without look-ahead we have the following accepting computation:

$$(p, abcba, \epsilon) \vdash (q, abcba, S) \vdash (q, abcba, aSa) \vdash (q, bcba, Sa) \vdash (q, bcba, bSba)$$

$$\vdash (q, cba, Sba) \vdash (q, cba, cba) \vdash (q, ba, ba) \vdash (q, a, a) \vdash (q, \epsilon, \epsilon)$$

This leads to acceptance. However, there are many other choices that do not lead to acceptance, causing inefficiency by the nondeterminism:

$$(p, abcba, \epsilon) \vdash (q, abcba, S) \vdash (q, abcba, bSb)$$

$$(p, abcba, \epsilon) \vdash (q, abcba, S) \vdash (q, abcba, c)$$

and there are choices like this later on, as well.

For the push-down automaton with look-ahead we have the following accepting computation:

$$(p, abcba\$, \epsilon) \vdash (q, abcba\$, S) \vdash (q_a, bcba\$, S) \vdash (q_a, bcba\$, aSa) \vdash (q, bcba\$, Sa)$$

$$\vdash (q_b, cba\$, Sa) \vdash (q_b, cba\$, bSba) \vdash (q, cba\$, Sba) \vdash (q_c, ba\$, Sba) \vdash (q_c, ba\$, cba)$$

$$\vdash (q, ba\$, ba) \vdash (q_b, a\$, ba) \vdash (q, a\$, a) \vdash (q_a, \$, a) \vdash (q, \$, \epsilon) \vdash (q_\$, \epsilon, \epsilon)$$

There are no choices, so that this push-down automaton is more efficient. Even for strings that are rejected, with look-ahead there is no nondeterminism.

- So, to get a deterministic push-down automaton, apply left-factoring, left recursion removal, and then one-symbol look-ahead.
- However, this won't always work, and it may be necessary to modify the grammar or the context-free language itself to get a deterministic push-down automaton.
- Grammars for which one-symbol look-ahead works, as presented here, are called $LL(1)$ grammars.
- From the run of the push-down automaton, one can construct a derivation and from it a parse tree that can be used for code generation.

1.10 Bottom-up parsing

There is another way to construct a push-down automaton for a context-free language, besides the method of lemma 3.4.1.

Given a context-free grammar $G = (V, \Sigma, R, S)$, one can construct a push-down automaton M such that $L(M) = L(G)$ as follows:

$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$ where Δ has the rules

(1)	$((p, \epsilon, S), (q, \epsilon))$	
(2)	$((p, \epsilon, \alpha^R), (p, A))$	if $A \to \alpha$ is in R (reduce the top of the stack, do a production in reverse)
(3)	$((p, a, \epsilon), (p, a))$	for each $a \in \Sigma$ (push (shift) input symbol on the stack)

- This push-down automaton attempts to construct a rightmost bottom-up derivation backwards from the given terminal string, to the start symbol S.

- Here is an example. Consider the grammar

$$S \to aSb, S \to cSd, S \to \epsilon.$$

The corresponding pda is this:

$M = (\{p, q\}, \{a, b, c, d\}, \{S, a, b, c, d\}, \Delta, p, \{q\})$ where Δ has the rules

(1)	$((p, \epsilon, S), (q, \epsilon))$	
(2)	$((p, \epsilon, bSa), (p, S))$	from $S \to aSb$
(2)	$((p, \epsilon, dSc), (p, S))$	from $S \to cSd$
(2)	$((p, \epsilon, \epsilon), (p, S))$	from $S \to \epsilon$
(3)	$((p, a, \epsilon), (p, a))$	for a
(3)	$((p, b, \epsilon), (p, b))$	for b
(3)	$((p, c, \epsilon), (p, c))$	for c
(3)	$((p, d, \epsilon), (p, d))$	for d

- Here is a computation: $(p, acdb, \epsilon) \vdash (p, cdb, a) \vdash (p, db, ca) \vdash (p, db, Sca) \vdash (p, b, dSca) \vdash (p, b, Sa) \vdash (p, \epsilon, bSa) \vdash (p, \epsilon, S) \vdash (q, \epsilon, \epsilon)$
- It is still nondeterministic, though; there is often a choice whether to *shift* (type 3 transition) or to *reduce* (type 2 transition). There also can be a conflict between type 2 transitions.
- A *precedence relation* is used to decide when to shift and when to reduce. When there is a choice of reductions, the longest one is chosen.
- This system works for the so-called *weak precedence* grammars, but does not work for all grammars.

1.11 Precedence Relations

- The *precedence relation* P is a subset of $V \times (\Sigma \cup \{\$\})$.
- Thus if $(a, b) \in V \times (\Sigma \cup \{\$\})$ then a is stack symbol and b is an input symbol or \$.

- Given the top-of-stack symbol a and the next input symbol b, if $(a, b) \in P$, this tells the push-down automaton to reduce (apply a production in reverse).
- If $(a, b) \notin P$, then this tells the push-down automaton to shift the next input symbol onto the stack.
- Such a precedence relation P can be found to make M a deterministic parser of $L(G)$ if G is a weak precedence grammar.
- There are systematic ways to compute the precedence relation P if G is a weak precedence grammar. This works, for example, for grammars for arithmetic expressions. For an example, see page 173 of the text.

The idea is that $(a, b) \in P$ means that there exists a rightmost derivation of the form

$$S \overset{R}{\Rightarrow}{}^*_G \beta Abx \overset{R}{\Rightarrow}_G \beta\gamma abx.$$

Here is an example of the precedence relation for the grammar for arithmetic expressions:

$$\begin{aligned} E &\to E + T \\ E &\to T \\ T &\to T * F \\ T &\to F \\ F &\to (E) \\ F &\to id \end{aligned}$$

Here is the table:

a \ b	(	)	id	+	$*$	\$
(						
)		x		x	x	x
id		x		x	x	x
+						
$*$						
E						
T		x		x		x
F		x		x	x	x

Here is the beginning of a computation for the string $id * (id)$ in this language (from the book, page 171):

$(p, id*(id), e) \vdash^s (p, *(id), id) \vdash^r (p, *(id), F) \vdash^r (p, *(id), T) \vdash^s (p, (id), *T) \vdash^s (p, id), (*T) \vdash \ldots$

Here $\vdash^s$ indicates a shift and $\vdash^r$ indicates a reduce operation of the push-down automaton.

As before, from an accepting computation of the push-down automaton, one can construct a derivation and then a parse tree that can be used for code genertion.

What happens if there is a mistake in the input string, so that the input string cannot be generated in the grammar? What will this parsing method do in such a case?