بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

Discovering a Derivation for an Input String

☞ We have seen how to use a CFG G as a rewrite system to generate sentences that are in $L(G)$. By contrast, a parser takes a given input string, alleged to be in $L(G)$, and finds a derivation. The process of constructing a derivation from a specific input sentence is called parsing.

☞ The parser sees the program as it emerges incrementally from the scanner: a stream of words annotated with their syntactic categories. Thus, the parser would see $a + b \cdot c$ as

$$\langle \mathbf{id}, a \rangle \langle + \rangle \langle \mathbf{id}, b \rangle \langle . \rangle \langle \mathbf{id}, c \rangle .$$

☞ As output, the parser needs to produce either a derivation of the input program or an error message that indicates an invalid program.

It is useful to visualize the parser as building a parse tree. The parse tree's root is known; it represents the grammar's start symbol. The leaves of the parse tree are known; they match the stream of words returned by the scanner. The hard part of parsing lies in finding the connection between the leaves and the root. Two distinct and opposite approaches for constructing the tree suggest themselves:

1. *Top-Down Parsers* begin with the root and grow the tree toward the leaves. At each step, a top-down parser selects a node for some nonterminal on the lower fringe of the partially built tree and extends it with a subtree that represents the right-hand side of a production that rewrites the nonterminal.
2. *Bottom-Up Parsers* begin with the leaves and grow the tree toward the root. At each step, a bottom-up parser finds a substring of the partially built parse tree's upper fringe that matches the right-hand side of some production; it then builds a node for the rule's left-hand side and connects it into the tree.

In either scenario, the parser makes a series of choices about which productions to apply. Most of the intellectual complexity in parsing lies in the mechanisms for making these choices.

Context-Free Grammars Versus Regular Expressions

Grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.

For example, the regular expression $(a|b)^*abb$ and the grammar

$$
\begin{aligned}
A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 &\rightarrow bA_2 \\
A_2 &\rightarrow bA_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

describe the same language, the set of strings of a's and b's ending in abb.

We can construct mechanically a grammar to recognize the same language as a nondeterministic finite automaton (NFA).

1. For each state i of the NFA, create a nonterminal A_i.
2. If state i has a transition to state j on input a, add the production $A_i \rightarrow aA_j$. If state i goes to state j on input ε, add the production $A_i \rightarrow A_j$.
3. If i is an accepting state, add $A_i \rightarrow \varepsilon$.
4. If i is the start state, make A_i be the start symbol of the grammar.

As we observed, everything that can be described by a regular expression can also be described by a grammar. We may therefore reasonably ask:

> *"Why use regular expressions to define the lexical syntax of a language?"*

There are several reasons.

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.

3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space. Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

Transformations

We consider several transformations that could be applied to get a grammar more suitable for parsing. One technique can eliminate ambiguity in the grammar, and other techniques — left-recursion elimination and left factoring — are useful for rewriting grammars so they become suitable for top-down parsing.

Rewriting a Grammar for $\mathrm{LL}(1)$ Parsing

Here, we will look at methods for rewriting grammars such that they are more palatable for $\mathrm{LL}(1)$ parsing. In particular, we will look at elimination of left-recursion and at left factorization.

It must, however, be noted that not all unambiguous grammars can be rewritten to allow $\mathrm{LL}(1)$ parsing. In these cases stronger parsing techniques must be used. We will not cover parsing of ambiguous grammars in this course.

Elimination of Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

Immediate (direct) left recursion: There is a production of the form $A \rightarrow A\alpha$.

The left-recursive pair of productions $A \rightarrow A\alpha|\beta$ could be replaced by the non-left-recursive productions:

$$A \rightarrow \beta A', \qquad A' \rightarrow \alpha A'|\varepsilon$$

without changing the strings derivable from A. This rule by itself suffices for many grammars.

Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

$$A \to A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$$

where no β_i begins with an A. Then, replace the A-productions by

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

The nonterminal A generates the same strings as before but is no longer left recursive.

This procedure eliminates all left recursion from the A and A' productions (provided no α_i is ε), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$
\begin{aligned}
S &\rightarrow A\ a \mid b \\
A &\rightarrow A\ c \mid S\ d \mid \epsilon
\end{aligned}
$$

The nonterminal S is left recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

Example

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

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow + T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow * F\ T' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Algorithm 4.19, below, systematically eliminates left recursion from a grammar. It is guaranteed to work if the grammar has no cycles (derivations of the form $A \Rightarrow^+ A$) or ε-productions (productions of the form $A \rightarrow \varepsilon$). Cycles can be eliminated systematically from a grammar, as can ε-productions (see Exercises 4.4.6 and 4.4.7).

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

Algorithm to eliminate left recursion from a grammar

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i-1$) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions
7) }

Figure 4.11: Algorithm to eliminate left recursion from a grammar

The procedure in Fig. 4.11 works as follows.

☞ *In the first iteration for $i = 1$, the outer for-loop of lines (2) through (7) eliminates any immediate left recursion among A_1-productions. Any remaining A_1-productions of the form $A_1 \rightarrow A_l\alpha$ must therefore have $l > 1$.*

☞ *After the $(i-1)$st iteration of the outer for-loop, all nonterminals A_k, where $k < i$, are "cleaned"; that is, any production $A_k \rightarrow A_l\alpha$, must have $l > k$.*

☞ *As a result, on the ith iteration, the inner loop of lines (3) through (5) progressively raises the lower limit in any production $A_i \rightarrow A_m\alpha$, until we have $m \geq i$. Then, eliminating immediate left recursion for the A_i productions at line (6) forces m to be greater than i.*

Example:

$$S \rightarrow A\ a \ \mid \ b$$
$$A \rightarrow A\ c \ \mid \ S\ d \ \mid \ \epsilon$$

Technically, the algorithm is not guaranteed to work, because of the ε-production, but in this case, the production $A \rightarrow \varepsilon$ turns out to be harmless. We order the nonterminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during the outer loop for $i = 1$.
For $i = 2$, we substitute for S in $A \rightarrow Sd$ to obtain the following A-productions: $A \rightarrow Ac|Aad|bd|\ \varepsilon$. Eliminating the immediate left recursion among these A-productions yields the following grammar.

$$S \rightarrow A\ a \ \mid \ b$$
$$A \rightarrow b\ d\ A' \ \mid \ A'$$
$$A' \rightarrow c\ A' \ \mid \ a\ d\ A' \ \mid \ \epsilon$$

Example (Removing Indirect Left Recursion): *Suppose in the following grammar that we want to remove the indirect left recursion that begins with the nonterminal A:*

$$A \to Bb|e, \qquad B \to Cc|f, \qquad C \to Ad|g.$$

During the outer loop for $i = 3$:
For $j = 1$:

$$C \to Bbd|ed|g$$

For $j = 2$:

$$C \to Ccbd|fbd|ed|g$$

The elimination of the immediate left recursion among the C-productions:

$$C \to fbdD|edD|gD, \qquad D \to cbdD|\varepsilon$$

Example (Cooper's Book)

Example

Consider the simple left-recursive grammar shown in the margin. Subscripts on the nonterminal symbols indicate the order used by the algorithm. The algorithm proceeds as follows:

$i = 0$	The algorithm does not enter the inner loop. No rule of the form $Goal_0 \rightarrow Goal_1\ \gamma$ found.
$i = 1$ $s = 0$	The algorithm looks for a rule with the form $A_1 \rightarrow Goal_0\ \gamma$. No such rule exists.
$i = 2$ $s = 0$	The algorithm looks for a rule with the form $B_2 \rightarrow Goal_0\ \gamma$. No such rule exists.
$i = 2$ $s = 1$	The algorithm looks for a rule with the form $B_2 \rightarrow A_1\ \gamma$. It finds rule 3 and rewrites it as $B_2 \rightarrow B_2$ a b | a b.

0	$Goal_0$	→	A_1
1	A_1	→	B_2 a
2		|	a
3	B_2	→	A_1 b

Original Grammar

0	$Goal_0$	→	A_1
1	A_1	→	B_2 a
2		|	a
3	B_2	→	a b C_3
4	C_3	→	a b C_3
5		|	ϵ

Transformed Grammar

Elimination of the ε-productions

! **Exercise 4.4.6:** A grammar is *ϵ-free* if no production body is ϵ (called an *ϵ-production*).

a) Give an algorithm to convert any grammar into an ϵ-free grammar that generates the same language (with the possible exception of the empty string — no ϵ-free grammar can generate ϵ). *Hint*: First find all the nonterminals that are *nullable*, meaning that they generate ϵ, perhaps by a long derivation.

b) Apply your algorithm to the grammar $S \rightarrow aSbS \mid bSaS \mid \epsilon$.

Elimination of the single productions

! **Exercise 4.4.7:** A *single production* is a production whose body is a single nonterminal, i.e., a production of the form $A \to B$.

a) Give an algorithm to convert any grammar into an ϵ-free grammar, with no single productions, that generates the same language (with the possible exception of the empty string) *Hint*: First eliminate ϵ-productions, and then find for which pairs of nonterminals A and B does $A \stackrel{*}{\Rightarrow} B$ by a sequence of single productions.

b) Apply your algorithm to the grammar (4.1) in Section 4.1.2.

c) Show that, as a consequence of part (a), we can convert a grammar into an equivalent grammar that has no *cycles* (derivations of one or more steps in which $A \stackrel{*}{\Rightarrow} A$ for some nonterminal A).