# Alcuni esercizi per II prova intercorso

I testi dei seguenti esercizi sono riportati nelle pagine successive.

pag. 241 n. 4.5.2 (Data la grammatica bisogna trovare l'handle in ciascuna forma sentenziale) , 4.5.3
pag. 258 n. 4.6.2, 4.6.5 e 4.6.6
pag. 277-278 n. 4.7.1, n. 4.7.2, 4.7.5 (almeno 4)
pag. 370 es. 6.2.1
pag 417 es. 6.7.1

Si estenda la grammatica S-Attribuita a pagina 377 (figura 6.46) del testo italiano per permettere la traduzione in codice a tre indirizzi della istruzione **repeat S while B** dove la condizione dopo il while deve essere falsa affichè si possa uscire dal repeat.

a) $S \rightarrow 0\ S\ 1 \mid 0\ 1$ with string 000111.

b) $S \rightarrow +\ S\ S \mid *\ S\ S \mid a$ with string $+ * aaa$.

! c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$ with string (()()).

! d) $S \rightarrow S\ +\ S \mid S\ S \mid (\ S\ ) \mid S\ * \mid a$ with string $(a + a) * a$.

! e) $S \rightarrow (\ L\ ) \mid a$ and $L \rightarrow L\ ,\ S \mid S$ with string $((a, a), a, (a))$.

!! f) $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$ with string $aabbab$.

! g) The following grammar for boolean expressions:

$$\begin{array}{lcl} \textit{bexpr} & \rightarrow & \textit{bexpr}\ \textbf{or}\ \textit{bterm} \mid \textit{bterm} \\ \textit{bterm} & \rightarrow & \textit{bterm}\ \textbf{and}\ \textit{bfactor} \mid \textit{bfactor} \\ \textit{bfactor} & \rightarrow & \textbf{not}\ \textit{bfactor} \mid (\ \textit{bexpr}\ ) \mid \textbf{true} \mid \textbf{false} \end{array}$$

**Exercise 4.2.3:** Design grammars for the following languages:

a) The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.

! b) The set of all strings of 0s and 1s that are *palindromes*; that is, the string reads the same backward as forward.

! c) The set of all strings of 0s and 1s with an equal number of 0s and 1s.

!! d) The set of all strings of 0s and 1s with an unequal number of 0s and 1s.

! e) The set of all strings of 0s and 1s in which 011 does not appear as a substring.

!! f) The set of all strings of 0s and 1s of the form $xy$, where $x \neq y$ and $x$ and $y$ are of the same length.

! **Exercise 4.2.4:** There is an extended grammar notation in common use. In this notation, square and curly braces in production bodies are metasymbols (like $\rightarrow$ or |) with the following meanings:

*i*) Square braces around a grammar symbol or symbols denotes that these constructs are optional. Thus, production $A \rightarrow X\ [Y]\ Z$ has the same effect as the two productions $A \rightarrow X\ Y\ Z$ and $A \rightarrow X\ Z$.

*ii*) Curly braces around a grammar symbol or symbols says that these symbols may be repeated any number of times, including zero times. Thus, $A \rightarrow X\ \{Y\ Z\}$ has the same effect as the infinite sequence of productions $A \rightarrow X$, $A \rightarrow X\ Y\ Z$, $A \rightarrow X\ Y\ Z\ Y\ Z$, and so on.

| | | | |
|---|---|---|---|
| (1) | *stmt* | → | **id** ( *parameter_list* ) |
| (2) | *stmt* | → | *expr* := *expr* |
| (3) | *parameter_list* | → | *parameter_list* , *parameter* |
| (4) | *parameter_list* | → | *parameter* |
| (5) | *parameter* | → | **id** |
| (6) | *expr* | → | **id** ( *expr_list* ) |
| (7) | *expr* | → | **id** |
| (8) | *expr_list* | → | *expr_list* , *expr* |
| (9) | *expr_list* | → | *expr* |

.30: Productions involving procedure calls and array references

| STACK | INPUT |
|---|---|
| ··· **id** ( **id** | , **id** ) ··· |

t that the **id** on top of the stack must be reduced, but by which The correct choice is production (5) if p is a procedure, but pro- if p is an array. The stack does not tell which; information in the e obtained from the declaration of p must be used.
tion is to change the token **id** in production (1) to **procid** and to sophisticated lexical analyzer that returns the token name **procid** gnizes a lexeme that is the name of a procedure. Doing so would exical analyzer to consult the symbol table before returning a token. de this modification, then on processing `p(i,j)` the parser would the configuration

| STACK | INPUT |
|---|---|
| ··· **procid** ( **id** | , **id** ) ··· |

nfiguration above. In the former case, we choose reduction by 5); in the latter case by production (7). Notice how the symbol he top of the stack determines the reduction to be made, even not involved in the reduction. Shift-reduce parsing can utilize far down in the stack to guide the parse. □

### ercises for Section 4.5

**5.1:** For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$ of Exercise 4.2.2(a), handle in each of the following right-sentential forms:

.

**5.2:** Repeat Exercise 4.5.1 for the grammar $S \rightarrow S\ S + \mid S\ S * \mid a$ .2.1 and the following right-sentential forms:



a) $SSS + a * +$.

b) $SS + a * a +$.

c) $aaa * a + +$.

**Exercise 4.5.3:** Give bottom-up parses for the following input strings and grammars:

a) The input 000111 according to the grammar of Exercise 4.5.1.

b) The input $aaa * a + +$ according to the grammar of Exercise 4.5.2.

## 4.6 Introduction to LR Parsing: Simple LR

The most prevalent type of bottom-up parser today is based on a concept called LR($k$) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the $k$ for the number of input symbols of lookahead that are used in making parsing decisions. The cases $k = 0$ or $k = 1$ are of practical interest, and we shall only consider LR parsers with $k \leq 1$ here. When ($k$) is omitted, $k$ is assumed to be 1.

This section introduces the basic concepts of LR parsing and the easiest method for constructing shift-reduce parsers, called "simple LR" (or SLR, for short). Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator. We begin with "items" and "parser states;" the diagnostic output from an LR parser generator typically includes parser states, which can be used to isolate the sources of parsing conflicts.

Section 4.7 introduces two, more complex methods — canonical-LR and LALR — that are used in the majority of LR parsers.

### 4.6.1 Why LR Parsers?

LR parsers are table-driven, much like the nonrecursive LL parsers of Section 4.4.4. A grammar for which we can construct a parsing table using one of the methods in this section and the next is said to be an *LR grammar*. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

! b) The grammar $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ of Exercise 4.2.1.

! c) The grammar $S \rightarrow S\ (\ S\ )\ |\ \epsilon$ of Exercise 4.2.2(c).

**Exercise 4.6.2:** Construct the SLR sets of items for the (augmented) grammar of Exercise 4.2.1. Compute the GOTO function for these sets of items. Show the parsing table for this grammar. Is the grammar SLR?

**Exercise 4.6.3:** Show the actions of your parsing table from Exercise 4.6.2 on the input $aa*a+$.

**Exercise 4.6.4:** For each of the (augmented) grammars of Exercise 4.2.2(a)–(g):

a) Construct the SLR sets of items and their GOTO function.

b) Indicate any action conflicts in your sets of items.

c) Construct the SLR-parsing table, if one exists.

**Exercise 4.6.5:** Show that the following grammar:

$$\begin{array}{lcl} S & \rightarrow & A\ a\ A\ b\ |\ B\ b\ B\ a \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

is LL(1) but not SLR(1).

**Exercise 4.6.6:** Show that the following grammar:

$$\begin{array}{lcl} S & \rightarrow & S\ A\ |\ A \\ A & \rightarrow & a \end{array}$$

is SLR(1) but not LL(1).

!! **Exercise 4.6.7:** Consider the family of grammars $G_n$ defined by:

$$\begin{array}{lcll} S & \rightarrow & A_i\ b_i & \text{for } 1 \leq i \leq n \\ A_i & \rightarrow & a_j\ A_i\ |\ a_j & \text{for } 1 \leq i, j \leq n \text{ and } i \neq j \end{array}$$

Show that:

a) $G_n$ has $2n^2 - n$ productions.

b) $G_n$ has $2^n + n^2 + n$ sets of LR(0) items.

c) $G_n$ is SLR(1).

What does this analysis say about how large LR parsers can get?



! **Exercise 4.6.8:** We sugg
states of a nondeterministic
states of a deterministic fin
an NFA" in Section 4.6.5).
Exercise 4.2.1:

a) Draw the transition d
according to the rule

b) Apply the subset cons
(a). How does the res
the grammar?

!! c) Show that in all cases
comes from the valid
items.

! **Exercise 4.6.9:** The follov

Construct for this grammar
build an LR-parsing table fo
What are they? Suppose w
tically choosing a possible
possible sequences of action

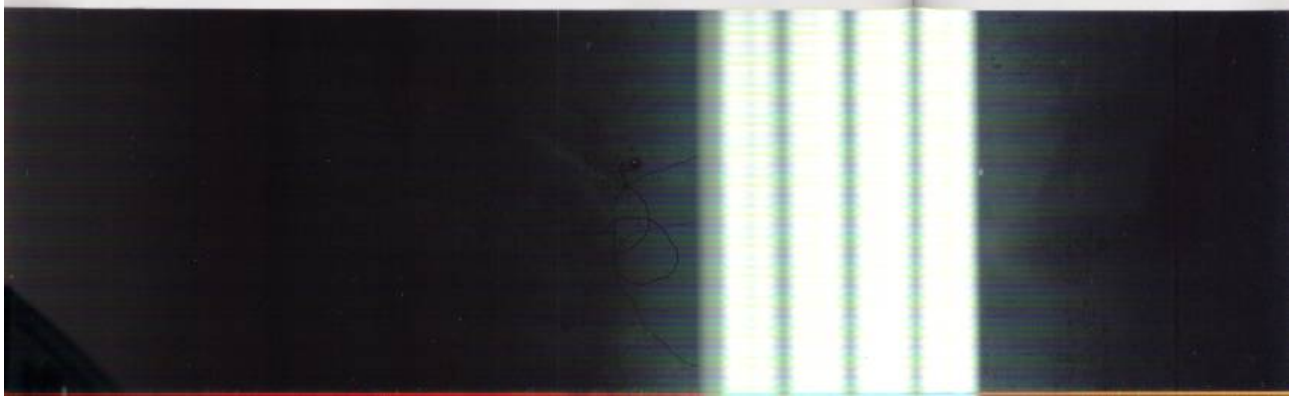
## 4.7 More Power

In this section, we shall ext
symbol of lookahead on the

1. The "canonical-LR"
lookahead symbol(s).
LR(1) items.

2. The "lookahead-LR"
sets of items, and has
LR(1) items. By care
we can handle many
the SLR method, and
SLR tables. LALR is

After introducing both thes
compact LR parsing tables



This technique is useful because there tend to be rather few states in any one column of the GOTO table. The reason is that the GOTO on nonterminal $A$ can only be a state derivable from a set of items in which some items have $A$ immediately to the left of a dot. No set has items with $X$ and $Y$ immediately to the left of a dot if $X \neq Y$. Thus, each state appears in at most one GOTO column.

For more space reduction, we note that the error entries in the goto table are never consulted. We can therefore replace each error entry by the most common non-error entry in its column. This entry becomes the default; it is represented in the list for each column by one pair with **any** in place of *currentState*.

**Example 4.66:** Consider Fig. 4.37 again. The column for $F$ has entry 10 for state 7, and all other entries are either 3 or error. We may replace error by 3 and create for column $F$ the list

| CURRENTSTATE | NEXTSTATE |
|---|---|
| 7 | 10 |
| **any** | 3 |

Similarly, a suitable list for column $T$ is

| | |
|---|---|
| 6 | 9 |
| **any** | 2 |

For column $E$ we may choose either 1 or 8 to be the default; two entries are necessary in either case. For example, we might create for column $E$ the list

| | |
|---|---|
| 4 | 8 |
| **any** | 1 |

□

This space savings in these small examples may be misleading, because the total number of entries in the lists created in this example and the previous one together with the pointers from states to action lists and from nonterminals to next-state lists, result in unimpressive space savings over the matrix implementation of Fig. 4.37. For practical grammars, the space needed for the list representation is typically less than ten percent of that needed for the matrix representation. The table-compression methods for finite automata that were discussed in Section 3.9.8 can also be used to represent LR parsing tables.

### 4.7.7 Exercises for Section 4.7

**Exercise 4.7.1:** Construct the

a) canonical LR, and

b) LALR

sets of items for the grammar $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ of Exercise 4.2.1.

**Exercise 4.7.2:** Repeat Exercise 4.7.1 for each of the (augmented) grammars of Exercise 4.2.2(a)–(g).

! **Exercise 4.7.3:** For the grammar of Exercise 4.7.1, use Algorithm 4.63 to compute the collection of LALR sets of items from the kernels of the LR(0) sets of items.

! **Exercise 4.7.4:** Show that the following grammar

$$\begin{array}{lcl} S & \rightarrow & A\ a\ |\ b\ A\ c\ |\ d\ c\ |\ b\ d\ a \\ A & \rightarrow & d \end{array}$$

is LALR(1) but not SLR(1).

! **Exercise 4.7.5:** Show that the following grammar

$$\begin{array}{lcl} S & \rightarrow & A\ a\ |\ b\ A\ c\ |\ B\ c\ |\ b\ B\ a \\ A & \rightarrow & d \\ B & \rightarrow & d \end{array}$$

is LR(1) but not LALR(1).

## 4.8 Using Ambiguous Grammars

It is a fact that every ambiguous grammar fails to be LR and thus is not in any of the classes of grammars discussed in the previous two sections. However, certain types of ambiguous grammars are quite useful in the specification and implementation of languages. For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar. Another use of ambiguous grammars is in isolating commonly occurring syntactic constructs for special-case optimization. With an ambiguous grammar, we can specify the special-case constructs by carefully adding new productions to the grammar.

Although the grammars we use are ambiguous, in all cases we specify disambiguating rules that allow only one parse tree for each sentence. In this way, the overall language specification becomes unambiguous, and sometimes it becomes possible to design an LR parser that follows the same ambiguity-resolving choices. We stress that ambiguous constructs should be used sparingly and in a strictly controlled fashion; otherwise, there can be no guarantee as to what language is recognized by a parser.



### 4.8.1 Preceden

Consider the ambiguc
*, repeated here for c

This grammar is amb
precedence of the ope
includes productions
but gives + lower prec
There are two reason
First, as we shall see,
of the operators + ar
number of states in t
biguous grammar will
productions $E \rightarrow T$ a
and precedence. The
time reducing by thes
a single nonterminal).

The sets of LR(0)
mented by $E' \rightarrow E$ ar
there will be parsing-
table from the sets of
$I_8$ generate these conf
the parsing action ta
$E \rightarrow E + E$ and shift
in FOLLOW($E$). Thus
similar conflict is gen
on inputs + and *. In
will generate these co

However, these pr
tivity information for
parser based on Fig. 4
the parser reaches a c

For convenience, the
shown under PREFIX.

If * takes precede
stack, preparing to re
This choice was made
grammar for the same
*, we know the parser

Here, $\phi(x_1, x_2)$ has the value $x_1$ if the control flow passes through the true part of the conditional and the value $x_2$ if the control flow passes through the false part. That is to say, the $\phi$-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the $\phi$-function.

### 6.2.5 Exercises for Section 6.2

**Exercise 6.2.1:** Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

**Exercise 6.2.2:** Repeat Exercise 6.2.1 for the following assignment statements:

*i.* `a = b[i] + c[j].`

*ii.* `a[i] = b*c - b*d.`

*iii.* `x = f(y+1) + 2.`

*iv.* `x = *p + &y.`

! **Exercise 6.2.3:** Show how to transform a three-address code sequence into one in which each defined variable gets a unique variable name.

## 6.3 Types and Declarations

The applications of types can be grouped under checking and translation:

- *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator. For example, the `&&` operator in Java expects its two operands to be booleans; the result is also of type boolean.
- *Translation Applications.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

2. *merge*$(p_1, p_2)$ concatenates the lists pointed to by $p_1$ and $p_2$, and returns a pointer to the concatenated list.

3. *backpatch*$(p, i)$ inserts $i$ as the target label for each of the instructions on the list pointed to by $p$.

### 6.7.2 Backpatching for Boolean Expressions

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker nonterminal $M$ in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

$B \rightarrow B_1 \;||\; M\; B_2 \mid B_1 \;\&\&\; M\; B_2 \mid !\; B_1 \mid (\; B_1\;) \mid E_1 \;\mathbf{rel}\; E_2 \mid \mathbf{true} \mid \mathbf{false}$
$M \rightarrow \epsilon$

The translation scheme is in Fig. 6.43.

1) $B \rightarrow B_1 \;||\; M\; B_2$ { *backpatch*($B_1$.*falselist*, $M$.*instr*);
*B.truelist* = *merge*($B_1$.*truelist*, $B_2$.*truelist*);
*B.falselist* = $B_2$.*falselist*; }

2) $B \rightarrow B_1 \;\&\&\; M\; B_2$ { *backpatch*($B_1$.*truelist*, $M$.*instr*);
*B.truelist* = $B_2$.*truelist*;
*B.falselist* = *merge*($B_1$.*falselist*, $B_2$.*falselist*); }

3) $B \rightarrow !\; B_1$ { *B.truelist* = $B_1$.*falselist*;
*B.falselist* = $B_1$.*truelist*; }

4) $B \rightarrow (\; B_1\;)$ { *B.truelist* = $B_1$.*truelist*;
*B.falselist* = $B_1$.*falselist*; }

5) $B \rightarrow E_1 \;\mathbf{rel}\; E_2$ { *B.truelist* = *makelist*(*nextinstr*);
*B.falselist* = *makelist*(*nextinstr* + 1);
*emit*('`if`' $E_1$.*addr* **rel**.*op* $E_2$.*addr* '`goto _`');
*emit*('`goto _`'); }

6) $B \rightarrow \mathbf{true}$ { *B.truelist* = *makelist*(*nextinstr*);
*emit*('`goto _`'); }

7) $B \rightarrow \mathbf{false}$ { *B.falselist* = *makelist*(*nextinstr*);
*emit*('`goto _`'); }

8) $M \rightarrow \epsilon$ { *M.instr* = *nextinstr*; }

Figure 6.43: Translation scheme for boolean expressions

Consider semantic action (1) for the production $B \rightarrow B_1 \;||\; M\; B_2$. If $B_1$ is true, then $B$ is also true, so the jumps on $B_1$.*truelist* become part of *B.truelist*. If $B_1$ is false, however, we must next test $B_2$, so the target for the jumps

If $S$ is the enclosing construct, then a break-statement is a jump to the first instruction after the code for $S$. We can generate code for the break by (1) keeping track of the enclosing statement $S$, (2) generating an unfilled jump for the break-statement, and (3) putting this unfilled jump on $S.nextlist$, where *nextlist* is as discussed in Section 6.7.3.

In a two-pass front end that builds syntax trees, $S.nextlist$ can be implemented as a field in the node for $S$. We can keep track of $S$ by using the symbol table to map a special identifier **break** to the node for the enclosing statement $S$. This approach will also handle labeled break-statements in Java, since the symbol table can be used to map the label to the syntax-tree node for the enclosing construct.

Alternatively, instead of using the symbol table to access the node for $S$, we can put a pointer to $S.nextlist$ in the symbol table. Now, when a break-statement is reached, we generate an unfilled jump, look up *nextlist* through the symbol table, and add the jump to the list, where it will be backpatched as discussed in Section 6.7.3.

Continue-statements can be handled in a manner analogous to the break-statement. The main difference between the two is that the target of the generated jump is different.

### 6.7.5 Exercises for Section 6.7

**Exercise 6.7.1:** Using the translation of Fig. 6.43, translate each of the following expressions. Show the true and false lists for each subexpression. You may assume the address of the first instruction generated is 100.

a) `a==b && (c==d || e==f)`

b) `(a==b || c==d) || e==f`

c) `(a==b && c==d) && e==f`

**Exercise 6.7.2:** In Fig. 6.47(a) is the outline of a program, and Fig. 6.47(b) sketches the structure of the generated three-address code, using the backpatching translation of Fig. 6.46. Here, $i_1$ through $i_8$ are the labels of the generated instructions that begin each of the "Code" sections. When we implement this translation, we maintain, for each boolean expression $E$, two lists of places in the code for $E$, which we denote by $E.true$ and $E.false$. The places on list $E.true$ are those places where we eventually put the label of the statement to which control must flow whenever $E$ is true; $E.false$ similarly lists the places where we put the label that control flows to when $E$ is found to be false. Also, we maintain for each statement $S$, a list of places where we must put the label to which control flows when $S$ is finished. Give the value (one of $i_1$ through $i_8$) that eventually replaces each place on each of the following lists:

(a) $E_3.false$ (b) $S_2.next$ (c) $E_4.false$ (d) $S_1.next$ (e) $E_2.true$