Building a Parser for ATC language

in the project seminar

Computational Natural Language Systems

RVS, Faculty of Technology, University of Bielefeld

Report RVS–Occ–01–05

Martin Ellermann[*] Mirco Hilbert[†]

February 18, 2002

Contents

1 Preface **1**

2 Generating the ATC Parser using Flex and Bison **1**

3 Generating the Flex and Bison input files **3**

3.1 The Flex input file 3

3.2 The Bison input file 4

3.3 Processing the ATC Grammar with the PERL program `compile.pl` 5

3.3.1 The main program `compile.pl` 5

3.3.2 The class SubstitutionTable 8

3.3.3 The class Rule 9

[*]`mellerma@TechFak.Uni-Bielefeld.de`

[†]`mail@Mirco-Hilbert.de`

3.3.4 The class EBNFRule 10
3.3.5 The class BisonRule 10

4 The generation process of the ATC Parser **13**

Bibliography **20**

A The Makefile to build `atc-parser.x` **21**

B The several steps when generating an ATC Parser **23**
B.1 An extract of `atc-grammar.txt` 23
B.2 An extract of `atc-grammar.ebnf` 25
B.3 An extract of `atc-grammar.sed` 27
B.4 The content of `atclib.h` 28
B.5 An extract of `atcscan.l` 28
B.6 An extract of `atcparse.y` 29
B.7 An extract of `atc.tab.h` 37

1 Preface

This document describes how we built a parser for a subset of grammatically well-defined ATC/pilot communications.

As described in [Hilb2001] we previously developed a machine-processable, descriptive grammar in EBNF syntax for a restricted sublanguage of ATC/pilot communications. The uniform declarative EBNF syntax allows relatively unrestricted machine processing and renders it independent of the parsing strategy a user may choose.

We chose a shift-reduce parsing strategy. To generate the parser we used the compiler development tools Flex and Bison[1]. To generate the input files needed by Flex and Bison we wrote a PERL program that takes the ATC Grammar as input.

The conversion algorithm that we implemented in this PERL program depends only on the EBNF syntax of the grammar and not on the ATC syntax the present grammar describes.

This makes the system modular with attendant benefits. When the grammar, on which the parser is based, is changed, only the conversion algorithm must be restarted to obtain a new parser.

2 Generating the ATC Parser using Flex and Bison

Flex generates C code for a lexical analyzer (*scanner*). It uses patterns that match strings in the input and converts the strings to tokens which are numerical representations of the strings. Bison generates C code for a syntax analyzer (*parser*). It uses grammar rules that allow it to analyze tokens from Flex.

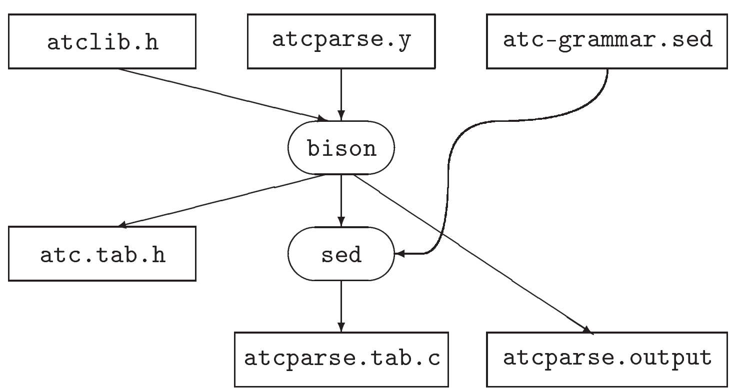


Figure 1: The process to generate the parser source file `atcparse.tab.c` using Bison.

[1]Flex and Bison are GNU ports of the tools Lex and Yacc by Lesk [Lesk1975] and Johnson [John1975]. A guide to Lex and Yacc is written by Thomas Niemann [Niem1997].

Bison needs as input a description of the grammar in a special syntax and a declaration of the tokens to be expected (contained in the file `atcparse.y`) (see figure 1)[2]. Besides the parser function `yyparse` (in the resulting file `atcparse.tab.c`) it generates the numerical constant definitions for every token declared in the input file (in the resulting file `atc.tab.h`).

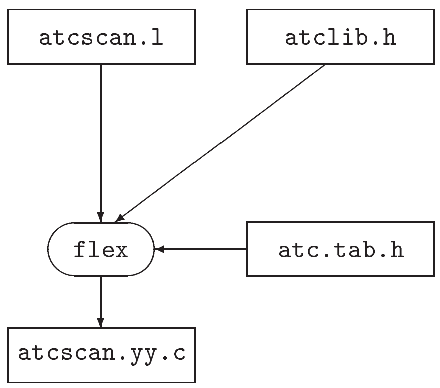


Figure 2: The process to generate the scanner source file `atcscan.yy.c` using Flex.

Flex needs as input a description of the string patterns (contained in the file `atcscan.l`) and includes the header file `atc.tab.h` with the constant definitions (see figure 2). It generates the scanner function `yylex` (in the resulting file `atcscan.yy.c`).

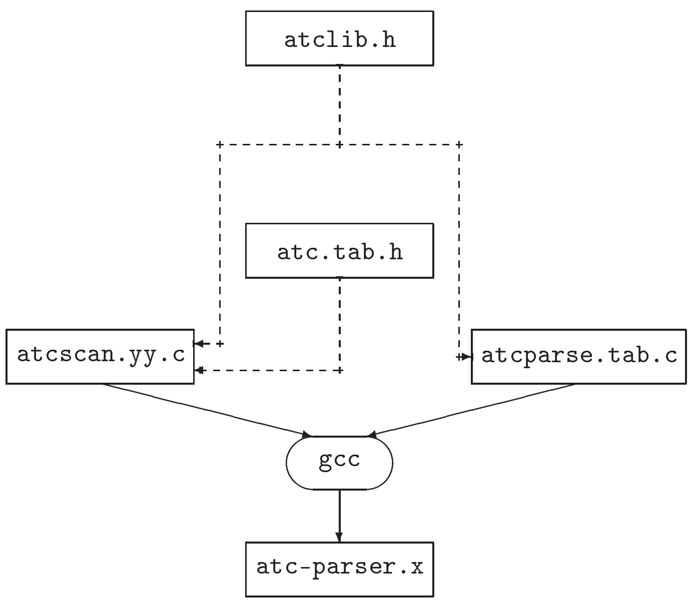


Figure 3: Compiling and linking the source files to generate the ATC parser `atc-parser.x`.

[2]All process diagrams and the legend for them in section 4 beginning on page 13.

The resulting C files are compiled and linked together to the executable parser file `atc-parser.x` (see figure 3). It is a shift-reduce parser consisting of the parser function `yyparse` which handles the *reduce* steps using the specified grammar rules and invoking the scanner function `yylex` when needed which *shift*s a new token from the input stream.

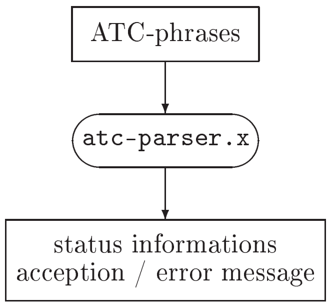


Figure 4: With the resulted executable file `atc-parser.x` you can parse ATC phrases.

3 Generating the Flex and Bison input files

When the ATC Parser is built using the tools Flex and Bison the task is now to generate the needed input files for Flex and Bison.

3.1 The Flex input file

The input file for Flex `atcscan.l` is divided into three parts.

The first part is optional and can include C macro definitions, declarations of variables, other C code and macro definitions for often used patterns, i. e. `letter    [A-Za-z]` defines the pattern `letter` which should stand for a capital letter from `A` to `Z` or a non-capital letter from `a` to `z`.

The second part includes the pattern rules where every pattern that should be matched in the input stream has a C code block allocated to it which will be executed when the pattern matches.

Example

`"BOSTON"    {RETURN(WORD_BOSTON);}` where

- `"BOSTON"` is the pattern string,
- `RETURN(...)` is a C macro which is defined in the first section and returns an integer value to the called parser function `yyparse`
- and `WORD_BOSTON` is an integer constant which is defined in the included header file `atc.tab.h`.

The third part includes C subroutines needed by the scanner.

While the first and the third part of the Flex input file must be written manualy the middle part can be automatically generated when a list of the occurring terminal strings is extracted from the present grammar. Then the generation of a pattern rule can be done by the following translation rule:

string $\Longrightarrow$ add the new pattern rule

"*string*" `{RETURN(`*token*`);}`

where *token* is `WORD_`*string*$'$ and *string*$'$ is an adjusted version of *string* where all special characters are replaced by underscores.

3.2 The Bison input file

The input file for Bison `atcparse.y` is also divided into three parts.

Besides C declarations the first part includes a list of tokens to be expected from the scanner.

The second part includes the grammar rules where each rule consists of a nonterminal on the left-hand side followed by a colon and one or more alternative strings of nonterminals and tokens divided by vertical lines on the right-hand side completed by a semicolon. Every alternative string can have in addition a C code block allocated to it which will be executed when the alternative is chosen.

Example

```
nonterminal_12 :
          WORD_1 WORD_2
        | WORD_TWELVE
        ;
```

The third part includes C subroutines needed by the parser and the main function that calls the parser.

Like the pattern rule part in the Lex input file the token section of the first part of the Bison input file can be generated automatically. The second part can be generated from the ATC Grammar when the following translation rules are implemented:

lhs `::=` *rhs* $\Longrightarrow$ add the new rule

$$lhs' \ \texttt{:}\ rhs' \ \texttt{;}$$

where lhs' is the recursively translated version of the left-hand side *lhs* and rhs' is the recursively translated version of the right-hand side *rhs*.

$\ldots\ string_1\ string_2\ \ldots$ $\Longrightarrow$ replace $string_1\ string_2$ by $string'_1\ string'_2$ where $string'_1$ is the recursively translated version of $string_1$ and $string'_2$ is the recursively translated version of $string_2$.

$\ldots\ string_1\ \texttt{|}\ string_2\ \ldots$ $\Longrightarrow$ replace $string_1\ \texttt{|}\ string_2$ by $string'_1\ \texttt{|}\ string'_2$ where $string'_1$ is the recursively translated version of $string_1$ and $string'_2$ is the recursively translated version of $string_2$.

$\ldots\ \texttt{(}string\texttt{)}\ \ldots$ $\Longrightarrow$ replace `(`*string*`)` by a new non-terminal *newsymb* and add a new rule

$$newsymb\ \texttt{:}\ string'\ \texttt{;}$$

where $string'$ is the recursively translated version of *string*.

$\ldots\ \texttt{[}string\texttt{]}\ \ldots$ $\Longrightarrow$ replace `[`*string*`]` by a new non-terminal *newsymb* and add a new rule

$$newsymb\ \texttt{:}\ string'\ \texttt{|}\ \texttt{/* empty */}\ \texttt{;}$$

where $string'$ is the recursively translated version of *string*.

$\ldots\ \texttt{\{}string\texttt{\}}\ \ldots$ $\Longrightarrow$ replace `{`*string*`}` by a new non-terminal *newsymb* and add a new rule

$$newsymb\ \texttt{:}\ string'\ \texttt{|}\ newsymb\ string'\ \texttt{;}$$

where $string'$ is the recursively translated version of *string*.

$\ldots\ \texttt{<}string\texttt{>}\ \ldots$ $\Longrightarrow$ replace `<`*string*`>` by the non-terminal *string*, possibly change *string* to Bison-syntax first.

$\ldots\ string\ \ldots$ $\Longrightarrow$ when *string* is only a single word replace *string* by its token *token* where *token* is $\texttt{WORD\_}string'$ and $string'$ is an adjusted version of *string* where all special characters are replaced by underscores.

3.3 Processing the ATC Grammar with the PERL program `compile.pl`

3.3.1 The main program `compile.pl`

The hole generation of the Lex and Bison input files `atcscan.l` and `atcparse.y` from the present ATC Grammar is managed by the PERL program `compile.pl` which uses some objects as instances of the classes which are described in the following.

compile.pl

```
// Global variables
terminalTable := a new instance of SubstitutionTable
nonterminalTable := a new instance of SubstitutionTable
ebnfRuleList := a new empty list
```

```
// Generate bisonTerminal from ebnfTerminal and
// insert the pair (ebnfTerminal, bisonTerminal) into the terminalTable
function insertTerminal(ebnfTerminal)
  if ebnfTerminal is a "." or a "," then
    bisonTerminal := "'.'" resp. "'.'"
  else if ebnfTerminal is a number then
    bisonTerminal := "WORD_" concatenated with ebnfTerminal
  else
    bisonTerminal := ebnfTerminal
    replace all non-capital letters in bisonTerminal by their corresponding capital letters
    replace all non-alphanumeric letters in bisonTerminal by underscores
    bisonTerminal := "WORD_" concatenated with bisonTerminal
  end if
  terminalTable.insert(ebnfTerminal, bisonTerminal)
end function

// Generate bisonNonterminal from ebnfNonterminal and
// insert the pair (ebnfNonterminal, bisonNonterminal) into the nonterminalTable
function insertNonterminal(ebnfNonterminal)
  bisonNonterminal := ebnfNonterminal
  erase the preceding "<" and the succeding ">" from bisonNonterminal
  if bisonNonterminal does not begin with a non-capital letter then
    bisonNonterminal := "nonterminal_" concatenated with bisonNonterminal
  end if
  nonterminalTable.insert(ebnfNonterminal, bisonNonterminal)
end function

// The compile process will be started by calling main(atc-grammar.txt)
// to generate the needed files atc-grammar.flex, atc-grammar.bison and atc-grammar.sed
function main(filename)
  grammar := read in the grammar file filename
```

erase all comment lines beginning with "`#`" from *grammar*
ruleList := split *grammar* into a list of individual rules

for each *rule* ∈ *ruleList* **do**
  lhs := left-hand side of *rule*
  rhs := right-hand side of *rule*
  erase linebreaks and unnecessary whitespaces from *rhs*
  write "*rhs* `::=` *lhs*" to the output file `atc-grammar.ebnf`
  insertNonterminal(*lhs*)
  ebnfRule := a new instance of EBNFRule with *lhs* on the left- and *rhs* on the right-hand side
  insert *ebnfRule* into *ebnfRuleList*
  extract all possible terminal strings from *rhs* and insert them into *terminalList*
end for

terminalList := sort *terminalList* while compacting multiple existing terminals to one
for each *terminal* ∈ *terminalList* **do**
  insertTerminal(*terminal*)
end for

// Generate the SEd script file `atc-grammar.sed`
write the string *terminalTable*.toInvertedSedString() to the output file `atc-grammar.sed`

// Generate the Flex pattern rules and write them to the file `atc-grammar.flex`
write the string *terminalTable*.toFlexString() to the output file `atc-grammar.flex`

// Generate the Bison token section and write it to the file `atc-grammar.bison`
tokens := concatenation of all substitutions of *terminalTable* divided by single spaces
write "`%token    EOI`" to the output file `atc-grammar.bison` // used for End Of Input
write "`%token    `" concatenated with *tokens* to the output file `atc-grammar.bison`
write "`%start    start`" to the output file `atc-grammar.bison` // defines the start rule
write an empty line, the string "`%%`" and a second empty line to the output file `atc-grammar.bison`
  to separate the first from the second part of the Bison input file

// Generate the Bison grammar rules and write them to the file `atc-grammar.bison`
write "
```
start :
                atc_block start     {}
              | EOI                 {printf("\n=> Input accepted.\n"); exit(0);}
              ;
```
" to the output file `atc-grammar.bison`
for each *ebnfRule* ∈ *ebnfRuleList* **do**
  bisonRuleSet := BisonRule.generateRuleSetFromEBNFRule(*ebnfRule*, *terminalTable*, *nonterminalTable*)
  for each *bisonRule* ∈ *bisonRuleSet* **do**

write the string *bisonRule*.toString() to the output file `atc-grammar.bison`
end for
end for
end function

3.3.2 The class SubstitutionTable

The class SubstitutionTable specifies an injective substitution set. That is, every key string in the first column of the table maps to an unique substitution in the second column. The class SubstitutionTable has two object instances in `compile.pl`: *terminalTable* and *nonterminalTable* to handle the translation of terminals (respectively non-terminals) between EBNF syntax and Bison syntax.

The method toFlexString() is used by the object *terminalTable* in `compile.pl` and implements – together with the method insert(*key*, *substitution*) and the function insertTerminal(*ebnfTerminal*) in `compile.pl` – the translation rule specified in 3.1 on page 4.

class SubstitutionTable

// Object variables
name := the name of the table // optional
table := a new empty hash table

// Insert the pair (*key*, *substitution*) in the actual table if it does not exist already in the table
// and if no other key exists with the same substitution
function insert(*key*, *substitution*)
if *key* is not defined or it exists already a substitution for *key* in the actual table **then**
throw an error message
else
if it exists a pair (*key′*, *substitution′*) in the actual table, where *substitution′* is the same as *substitution* **then**
substitution := *substitution* concatenated with `"_"` and a serial number, so that the new substitution does not exist in the actual table
end if
insert the pair (*key*, *substitution*) into the actual table
end if
end function

```
// Replace all keys that occur in string by their corresponding substitutions and return the
// resulting string
function substitute(string)
  for each key ∈ list of keys of the actual table do
    string := string where key is replaced by the substitution of key
  end for
  return string
end function

function toInvertedSedString()
  string := a new empty string
  for each key ∈ alphabetical ordered list of keys of the actual table do
    substitution := the substitution of key
    if substitution begins with "WORD_" then
      string := string concatenated with "s/"substitution"/"word key"/"
    else if substitution is "'.'" or "'.'" then
      string := string concatenated with "s/"substitution"/"character key"/"
    else
      string := string concatenated with "s/"substitution"/"input key"/"
    end if
  end for
  return string
end function

function toFlexString()
  string := a new empty string
  for each key ∈ alphabetical ordered list of keys of the actual table do
    substitution := the substitution of key
    string := string concatenated with ""key"    {column += yyleng; RETURN(substitution);}"
  end for
  return string
end function
```

3.3.3 The class Rule

The class Rule specifies an abstract grammar rule with a left-hand side as the name of the rule and a right-hand side and defines some classification constants for rules. It is the super class for EBNFRule and BisonRule which specify grammar rules of a definite syntax.

class Rule

// Global constants
// Numerical type constants for the type of a rule which indicates whether the rule describes a
// token, a sentence or a block or whether it is automatically generated, that is if it is a generated
// group, optional or recursion rule.

UNDEFINED := 0 // rule type is not defined (default type)
TOKEN := 10 // token rule
BLOCK := 20 // block rule
SENTENCE := 30 // sentence rule
GENERATED := 50 // generated rule
GENERATED_GRP := 51 // generated group rule
GENERATED_OPT := 52 // generated optional rule
GENERATED_REC := 53 // generated recursion rule

// Object variables

name := the name of the rule // corresponds to the left-hand side of the rule
rhs := the right-hand side of the rule
type := the type of the rule

3.3.4 The class EBNFRule

The class EBNFRule specifies a grammar rule in EBNF syntax.

class EBNFRule **extends** Rule

// Object variables inherited from Rule

name := the name of the rule // corresponds to the left-hand side of the rule
rhs := the right-hand side of the rule
type := the type of the rule

3.3.5 The class BisonRule

The class BisonRule specifies a grammar rule in Bison syntax.

It provides the public static method generateRuleSetFromEBNFRule(*ebnfRule*, *terminalTable*, *nonterminalTable*) which takes a rule in EBNF syntax as input and generates an equivalent set of rules in Bison syntax. This method implements – together with the method generateRule(*name*, *rhs*, *type*) – the translation rules specified in 3.2 on page 5.

class BisonRule **extends** Rule

// Object variables inherited from Rule

name := the name of the rule // corresponds to the left-hand side of the rule

rhs := the right-hand side of the rule

type := the type of the rule

// Returns an instance of BisonRule with a right-hand side that belongs to the specified *type*
// and is generated of *rhs*

function generateRule(*name*, *rhs*, *type*)
  if *type* is undefined **then**
    type := *GENERATED*
  end if
  rule := a new instance of BisonRule with the name *name* and the type *type*
  if *type* is *GENERATED_GRP* **then**
    set the right-hand side of *rule* to *rhs*
  else if *type* is *GENERATED_OPT* **then**
    set the right-hand side of *rule* to "*rhs* `| /* empty */`"
  else if *type* is *GENERATED_REC* **then**
    set the right-hand side of *rule* to "*rhs* `|` *name* *rhs*"
  else
    set the right-hand side of *rule* to *rhs*
  end if
  return *rule*
end function

function generateRuleSetFromEBNFRule(*ebnfRule*, *terminalTable*, *nonterminalTable*)
  bisonRuleSet := a new empty set

  // generate an instance of BisonRule with a syntactical illegal right-hand side
  rule := BisonRule.generateRule(name of *ebnfRule*, rhs of *ebnfRule*, type of *ebnfRule*)
  searchfurther := true
  while *searchfurther* **do**
    rhs := rhs of *rule*
    grp := the shortest but not empty string between a "`(`" and a "`)`" in *rhs*
    opt := the shortest but not empty string between a "`[`" and a "`]`" in *rhs*
    rec := the shortest but not empty string between a "`{`" and a "`}`" in *rhs*
    if at least one of *grp*, *opt* and *rec* was found **then**
      if the string *grp* is the shortest of *grp*, *opt* and *rec* **then**
        newname := the name of *rule* concatenated with "`_grp`" and a serial number

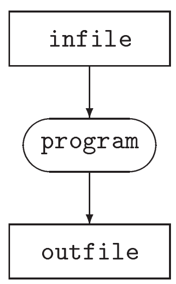
```
        nonterminalTable.insert("<newname>", newname)
        innerrule := BisonRule.generateRule(newname, grp, GENERATED_GRP)
        insert innerrule into bisonRuleSet
        rhs := rhs where "(grp)" is replaced by "<newname>"
        rule := BisonRule.generateRule(name of rule, rhs, type of rule)
      else if the string opt is the shortest of grp, opt and rec then
        newname := the name of rule concatenated with "_opt" and a serial number
        nonterminalTable.insert("<newname>", newname)
        innerrule := BisonRule.generateRule(newname, opt, GENERATED_OPT)
        insert innerrule into bisonRuleSet
        rhs := rhs where "[opt]" is replaced by "<newname>"
        rule := BisonRule.generateRule(name of rule, rhs, type of rule)
      else if the string rec is the shortest of grp, opt and rec then
        newname := the name of rule concatenated with "_rec" and a serial number
        nonterminalTable.insert("<newname>", newname)
        innerrule := BisonRule.generateRule(newname, rec, GENERATED_REC)
        insert innerrule into bisonRuleSet
        rhs := rhs where "{rec}" is replaced by "<newname>"
        rule := BisonRule.generateRule(name of rule, rhs, type of rule)
      else
        throw an error message
      end if
    else
      insert rule into bisonRuleSet
      searchfurther := false
    end if
  end while
  for each rule ∈ bisonRuleSet do
    rhs := rhs of rule
    rhs := terminalTable.substitute(rhs)
    rhs := nonterminalTable.substitute(rhs)
    change rhs of rule to rhs
  end for
  return bisonRuleSet
end function

// Return a string representation of the actual rule in Bison syntax
function toString()
```

```
  name := the name of the actual rule
  rhs := the rhs of the actual rule
  if the type of the actual rule is BLOCK or SENTENCE then
    typename := the name of the type of the actual rule
    rhs := rhs where the string "{printf("\n >> Instance of typename-rule <name> accepted.");}"
          is inserted after each alternative of rhs that is devided by a vertical line
    return "name : rhs ;"
  else
    return "name : rhs ;"
  end if
end function
```

4 The generation process of the ATC Parser illustrated by process diagrams

Input or output data stored in the file `file`.

An executable program stored in the file `program`.



Processing the program `program` with the file `infile` as input and producing the output file `outfile`.

Figure 5: The legend for the following process diagramms.

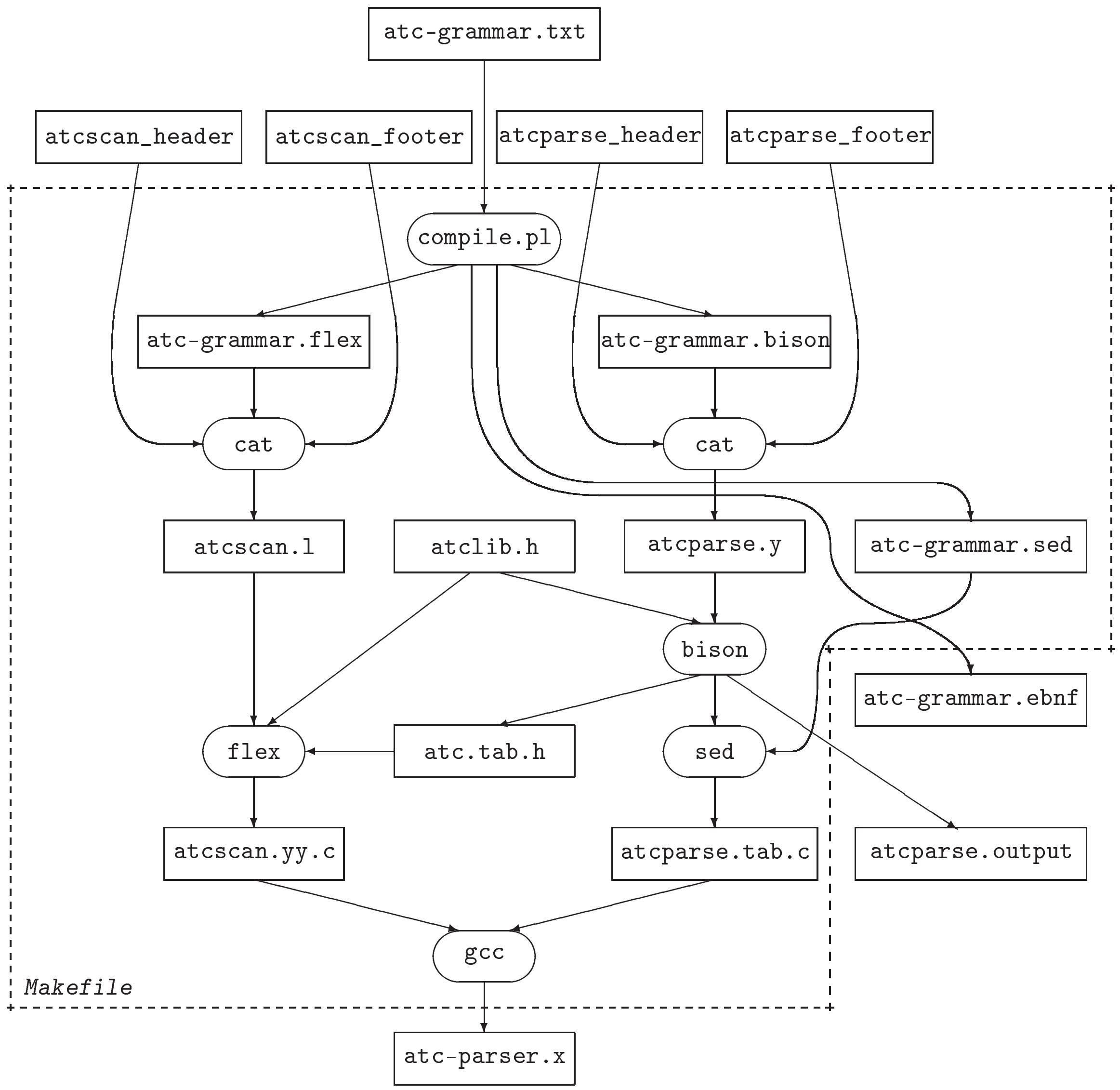


Figure 6: The whole process to generate the final executable ATC Parser `atc-parser.x` is managed by a Makefile.

This diagram is splitted in two main logical parts in the following figures 7 and 8, which are again splitted in several subparts in the figures 9 to 13.

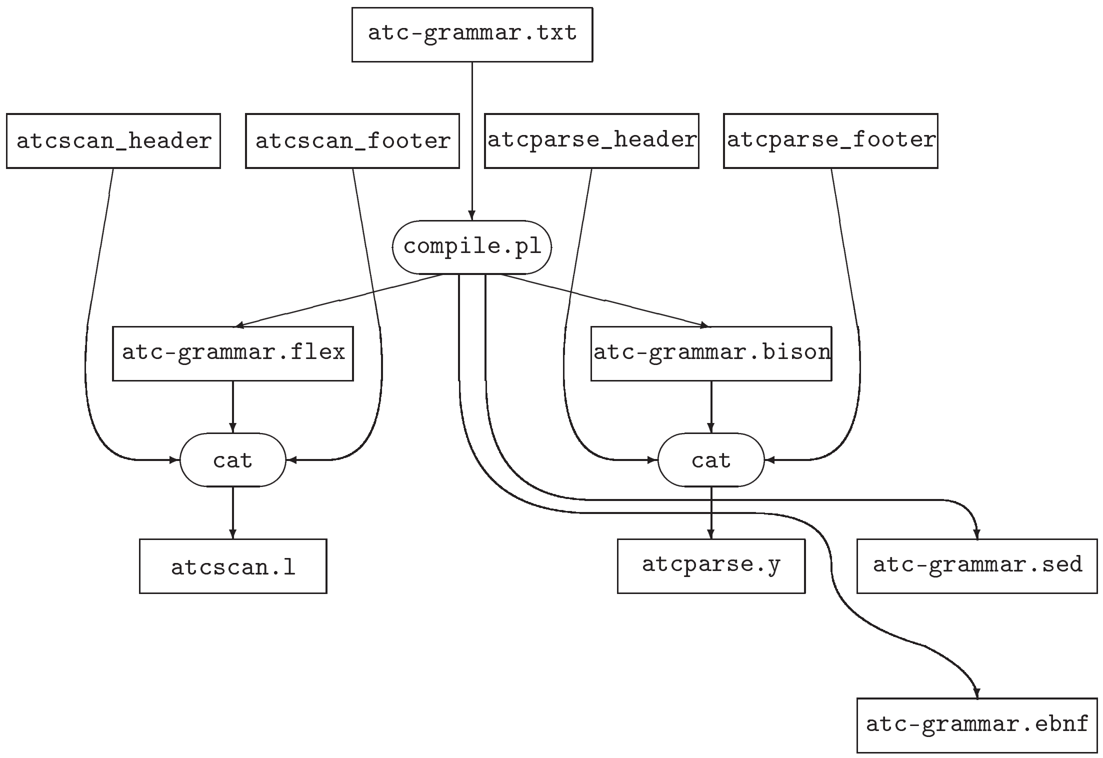


Figure 7: The process to generate the Flex input file `atcscan.l`, the Bison input file `atcparse.y` and the SEd input file `atc-grammar.sed`.
This diagram is splitted in logical parts in the figures 9 and 10.

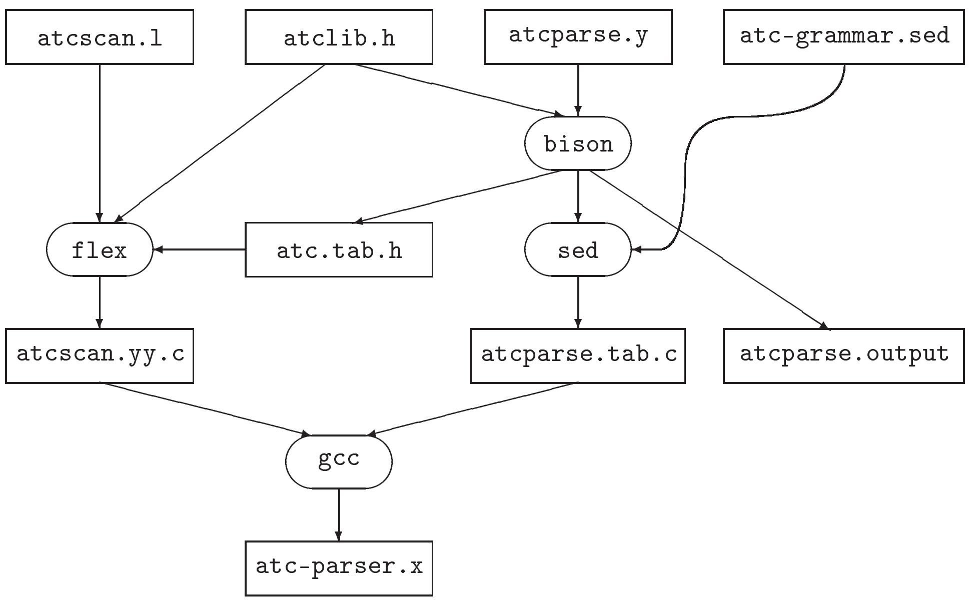


Figure 8: The process to generate the ATC Parser `atc-parser.x` out of the Flex and Bison input files `atcscan.l` and `atcparse.y`.
This diagram is splitted in logical parts in the figures 11, 12 and 13.

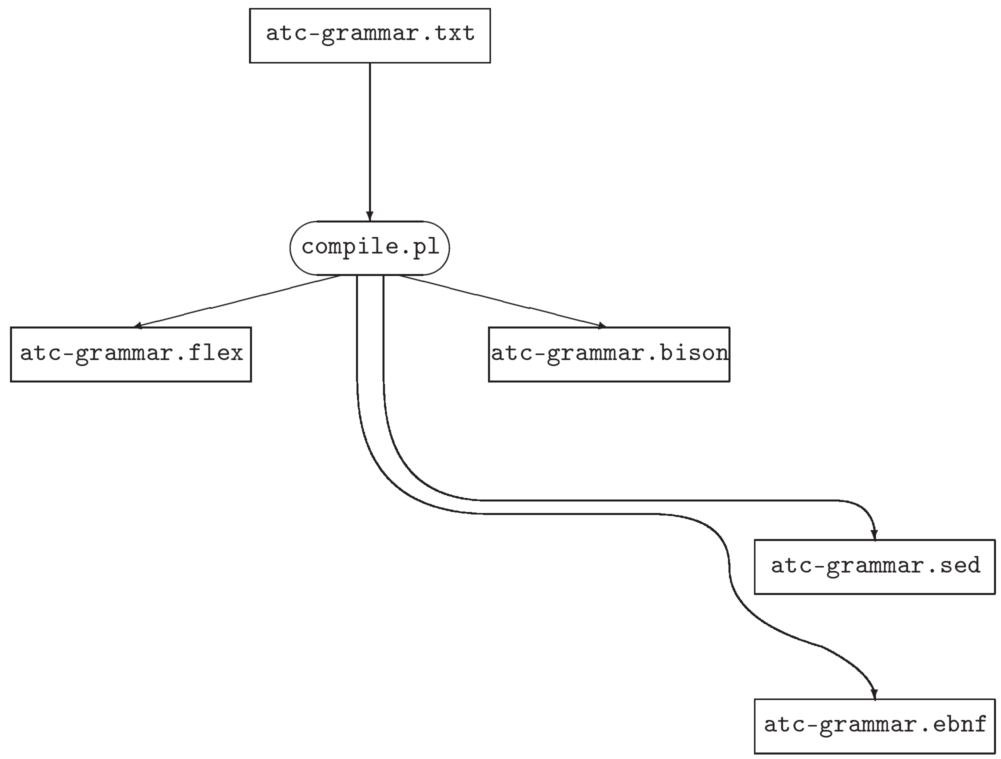


Figure 9: Running the PERL program `compile.pl` (decribed in 3.3) which reads in the original declarative ATC Grammar.

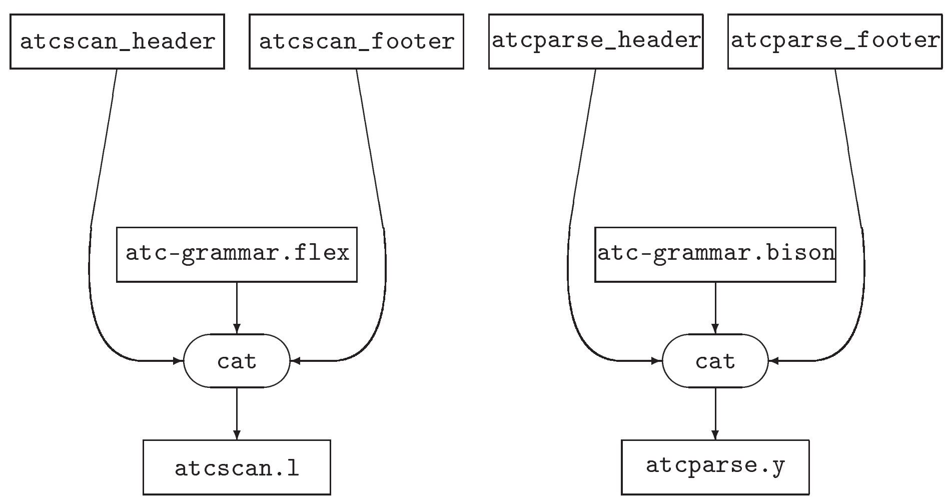


Figure 10: Concatenating the flex source file with its header and footer and analogous the bison source file.

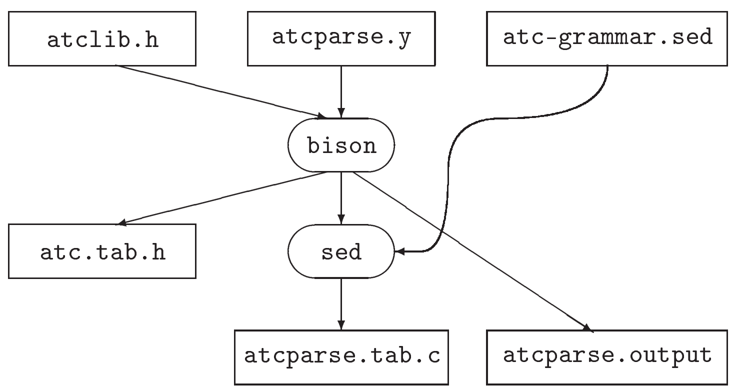


Figure 11: The process to generate the parser source file `atcparse.tab.c` using Bison.

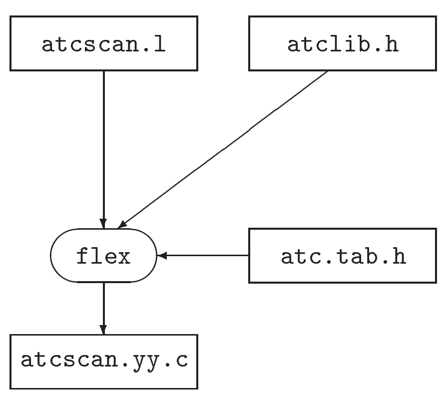


Figure 12: The process to generate the scanner source file `atcscan.yy.c` using Flex.

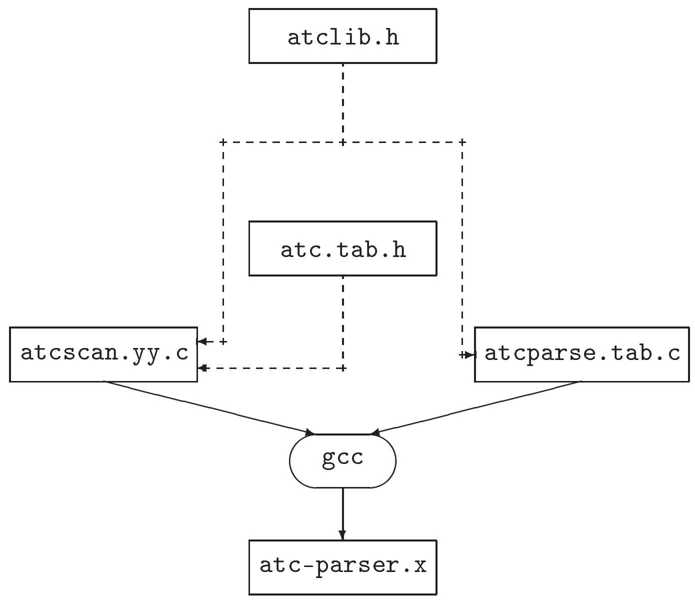


Figure 13: Compiling and linking the source files to generate the ATC parser `atc-parser.x`.

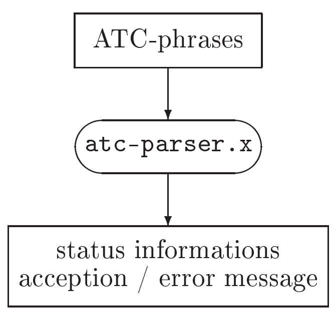


Figure 14: With the resulted executable file `atc-parser.x` you can parse ATC phrases.

Bibliography

References

[Elle2001] Martin Ellermann, Mirco Hilbert,
Review of the Cushing Grammar,
Technical Report RVS–Occ–01–02, RVS Group, Faculty of Technology,
University of Bielefeld, 2001

[Hilb2001] Mirco Hilbert, Martin Ellermann,
Developing an ATC Grammar using the Review of the Cushing Grammar,
Technical Report RVS–Occ–01–03, RVS Group, Faculty of Technology,
University of Bielefeld, 2001

[Niem1997] Thomas Niemann, *A Guide to Lex & Yacc*,
Portland, Oregon, 1997,
`http://epaperpress.com/lexandyacc/`

Resources

[John1975] Stephen C. Johnson,
Yacc: Yet Another Compiler Compiler,
Computing Science Technical Report No. 32, Bell Laboratories,
Murray Hill, New Jersey, 1975

[Lesk1975] M. E. Lesk, E. Schmidt,
Lex – A Lexical Analyzer Generator,
Computing Science Technical Report No. 39, Bell Laboratories,
Murray Hill, New Jersey, 1975

A The Makefile to build atc-parser.x

```
COMPILERDIR=        EBNF-Compiler


all:        atc-parser.x


atc-grammar.flex:        atc-grammar.txt $(COMPILERDIR)/compile.pl
        @echo ż Compiling atc-grammar.txt ...
        @(cd $(COMPILERDIR)/; ./compile.pl ../atc-grammar)
        @echo


atc-grammar.bison:        atc-grammar.txt $(COMPILERDIR)/compile.pl
        @echo ż Compiling atc-grammar.txt ...
        @(cd $(COMPILERDIR)/; ./compile.pl ../atc-grammar)
        @echo


atc-grammar.sed:        atc-grammar.txt $(COMPILERDIR)/compile.pl
        @echo ż Compiling atc-grammar.txt ...
        @(cd $(COMPILERDIR)/; ./compile.pl ../atc-grammar)
        @echo


atcscan.l:        atcscan_header atc-grammar.flex atcscan_footer
        @echo ż Erstellen von $@ aus atcscan_header, atc-grammar.flex und atcscan_footer ...
        @cat atcscan_header atc-grammar.flex atcscan_footer > $@
        @echo


atcparse.y:        atcparse_header atc-grammar.bison atcparse_footer
        @echo ż Erstellen von $@ aus atcparse_header, atc-grammar.bison und atcparse_footer ...
        @cat atcparse_header atc-grammar.bison atcparse_footer > $@
        @echo


atcparse.tab.c:        atcparse.y atc-grammar.sed atclib.h
        @echo ż Erstellen von atcparse.tab.c atc.tab.h atcparse.output ...
        @bison -d -v -o parse.tmp atcparse.y
        @sed -f atc-grammar.sed parse.tmp > $@
        @mv parse.tmp.h atc.tab.h
```

```
	@mv parse.tmp.output atcparse.output
	@rm -f parse.tmp
	@echo


atc.tab.h:	atcparse.tab.c


atcscan.yy.c:	atcscan.l atc.tab.h atclib.h
	@echo ż Erstellen von atcscan.yy.c ...
	@flex -o$@ atcscan.l
	@echo

atc-scanner.x:	atcscan.yy.c
	@echo ż Erstellen von atc-scanner.x ...
	@gcc -o $@ atcscan.yy.c
	@chmod ug+x $@
	@echo

atc-parser.x:	atcparse.tab.c atcscan.yy.c
	@echo ż Erstellen von atc-parser.x ...
	@gcc -o $@ atcscan.yy.c atcparse.tab.c
	@chmod ug+x $@
	@echo
	@echo Parsen einer Datei mit:  "cat <dateiname> | atc-parser.x"
	@echo

clean:
	@echo ż Cleaning ...
	@rm -fv atc-grammar.terminals atc-grammar.non-terminals atc-grammar.ebnf
	@rm -fv atc-grammar.flex atc-grammar.bison atc-grammar.sed
	@rm -fv atcscan.l atcparse.y
	@rm -fv atcparse.tab.c atc.tab.h atcparse.output atcscan.yy.c
	@rm -fv parse.tmp*
	@echo

clean_all:	clean
	@rm -fv atc-parser.x
	@echo
```

B The several steps when generating an ATC Parser from ATC Grammar version 1.0

B.1 An extract of atc-grammar.txt

(For the complete ATC Grammar see [Hilb2001].)

```
# Several information about the different token definitions are from the
# "Aeronautic Information Manual" (AIM), chapter 4 "Air Traffic Control",
# section 2 "Radio Communications Phraseology and Techniques"
# or from the "Air Traffic Control" handbook (ATC)
# of the "American Federal Aviation Administration Academy" (FAA)
# ( http://www.faa.gov/ATPubs/AIM/ resp. http://www.faa.gov/ATPubs/ATC/ )


<0>                 0 | ZERO
<1>                 1 | ONE
<2>                 2 | TWO
<3>                 3 | THREE
<4>                 4 | FOUR
<5>                 5 | FIVE
<6>                 6 | SIX
<7>                 7 | SEVEN
<8>                 8 | EIGHT
<9>                 9 | NINE | NINER | NINA
<10>                1 0 | TEN
<11>                1 1 | ELEVEN
<12>                1 2 | TWELVE

<point>             . | POINT


# Positiv digits (without 0)
<pos_digit>         <1> | <2> | <3> | <4> | <5> | <6> | <7> | <8> | <9>


# All digits (with 0)
<digit>             <0> | <pos_digit>

<digit_seq>         {<digit>}


# Numbers without leading zeros
```

```
<number>            <0> | <pos_digit> [<digit_seq>]


# Numbers without leading zeros greater than one
<number_gt_one>     <2> | <3> | <4> | <5> | <6> | <7> | <8> | <9> |
                    <pos_digit> <digit_seq>
```

[. . .]

```
# Contact instruction
# flightphase: all
# category:    radio operations
<block02>      <sentence03>

<sentence03>   CONTACT (<fname> | <lname>) <ffunction> [<frequency>]
               [AT (<time> | <fix> | <altitude>)].


# Frequency setting
# flightphase: all
# category:    radio operations
<block03>      <sentence04>

<sentence04>   CHANGE TO MY FREQUENCY <frequency>.


# Frequency setting
# flightphase: all
# category:    radio operations
<block04>      <sentence05>

<sentence05>   REMAIN THIS FREQUENCY.


# Traffic
# flightphase: cruise
# category:    other planes
<block05>      <sentence06>

<sentence06>   TRAFFIC, (
                  <clock_az> |
                  <direction>, <miles>, [<quad> BOUND,] <rel_movement>,
                  [<craft_type>,] (
```

```
                    <altitude> |
                    ALTITUDE UNKNOWN
                 )
              ).


# Traffic
# flightphase: cruise
# category:    other planes
<block06>      <block06a> | <block06b>

<block06a>     <sentence07>

<sentence07>   TRAFFIC, (<miles> | <minutes> MINUTES) <direction> OF
               (<fname> | <fix>), <direction> BOUND, [<craft_type>,]
               (<altitude> | ALTITUDE UNKNOWN).

<block06b>     <sentence08>

<sentence08>   TRAFFIC, NUMEROUS TARGETS VICINITY (<fname> | <fix>).
```

[. . .]

```
# Altitude setting, speed setting
# flightphase: cruise
# category:    altitude, speed
<block51>      <sentence85>

<sentence85>   MAINTAIN (<altitude> | <block_altitude>) [AT <speed>].


# Speed setting
# flightphase: cruise
# category:    speed
<block52>      <sentence86>

<sentence86>   RESUME NORMAL SPEED.
```

B.2 An extract of `atc-grammar.ebnf`

```
<0>             ::= 0 | ZERO

<1>             ::= 1 | ONE
```

```
<2>            ::= 2 | TWO

<3>            ::= 3 | THREE

<4>            ::= 4 | FOUR

<5>            ::= 5 | FIVE

<6>            ::= 6 | SIX

<7>            ::= 7 | SEVEN

<8>            ::= 8 | EIGHT

<9>            ::= 9 | NINE | NINER | NINA

<10>           ::= 1 0 | TEN

<11>           ::= 1 1 | ELEVEN

<12>           ::= 1 2 | TWELVE

<point>        ::= . | POINT

<pos_digit>    ::= <1> | <2> | <3> | <4> | <5> | <6> | <7> | <8> | <9>

<digit>        ::= <0> | <pos_digit>

<digit_seq>    ::= {<digit>}

<number>       ::= <0> | <pos_digit> [<digit_seq>]

<number_gt_one> ::= <2> | <3> | <4> | <5> | <6> | <7> | <8> | <9> | <pos_digit> <digit_seq>
```

[. . .]

```
<block02>      ::= <sentence03>

<sentence03>   ::= CONTACT (<fname> | <lname>) <ffunction> [<frequency>] [AT (<time> | <fix> | <altitude>)].

<block03>      ::= <sentence04>
```

```
<sentence04>    ::= CHANGE TO MY FREQUENCY <frequency>.

<block04>       ::= <sentence05>

<sentence05>    ::= REMAIN THIS FREQUENCY.

<block05>       ::= <sentence06>

<sentence06>    ::= TRAFFIC, ( <clock_az> | <direction>, <miles>, [<quad> BOUND,] <rel_movement>, [<craft_type>,] ( <altitude> | ALTITUDE UNKNOWN ) ).

<block06>       ::= <block06a> | <block06b>

<block06a>      ::= <sentence07>

<sentence07>    ::= TRAFFIC, (<miles> | <minutes> MINUTES) <direction> OF (<fname> | <fix>), <direction> BOUND, [<craft_type>,] (<altitude> | ALTITUDE UNKNOWN).

<block06b>      ::= <sentence08>

<sentence08>    ::= TRAFFIC, NUMEROUS TARGETS VICINITY (<fname> | <fix>).
```

[. . .]

```
<block51>       ::= <sentence85>

<sentence85>    ::= MAINTAIN (<altitude> | <block_altitude>) [AT <speed>].

<block52>       ::= <sentence86>

<sentence86>    ::= RESUME NORMAL SPEED.
```

B.3 An extract of `atc-grammar.sed`

```
s/\"WORD_TRANSPONDER_EQUIPPED\"/\"word TRANSPONDER-EQUIPPED\"/
s/\"WORD_MALFUNCTIONING\"/\"word MALFUNCTIONING\"/
s/\"WORD_INTERNATIONAL\"/\"word INTERNATIONAL\"/
s/\"WORD_INTERCEPTING\"/\"word INTERCEPTING\"/
s/\"WORD_DIAPERBLEACH\"/\"word DIAPERBLEACH\"/
s/\"WORD_TRANSPONDER\"/\"word TRANSPONDER\"/
s/\"WORD_INOPERATIVE\"/\"word INOPERATIVE\"/
s/\"WORD_ESTABLISHED\"/\"word ESTABLISHED\"/
s/\"WORD_IMMEDIATELY\"/\"word IMMEDIATELY\"/
s/\"WORD_CENTERFIELD\"/\"word CENTERFIELD\"/
```

[. . .]

```
s/\"WORD_Z\"/\"word Z\"/
s/\"WORD_3\"/\"word 3\"/
s/\"WORD_P\"/\"word P\"/
```

B.4 The content of `atclib.h`

```
int line, column;
```

B.5 An extract of `atcscan.l`

```
%{
#include "atc.tab.h"
#include "atclib.h"
#include <string.h>

#define DEBUGMODE 1
#define RETURN(V) {
  if (DEBUGMODE) {
    printf("return token %s\tin line %d, column %d\n", #V, line, column);
  }
  return V;
}
%}


whtspc                  [ \t]
word                    [A-Z_]+

%%

"TRANSPONDER-EQUIPPED"  {column += yyleng; RETURN(WORD_TRANSPONDER_EQUIPPED);}
"MALFUNCTIONING"        {column += yyleng; RETURN(WORD_MALFUNCTIONING);}
"INTERNATIONAL"         {column += yyleng; RETURN(WORD_INTERNATIONAL);}
"INTERCEPTING"          {column += yyleng; RETURN(WORD_INTERCEPTING);}
"DIAPERBLEACH"          {column += yyleng; RETURN(WORD_DIAPERBLEACH);}
"TRANSPONDER"           {column += yyleng; RETURN(WORD_TRANSPONDER);}
"INOPERATIVE"           {column += yyleng; RETURN(WORD_INOPERATIVE);}
"ESTABLISHED"           {column += yyleng; RETURN(WORD_ESTABLISHED);}
"IMMEDIATELY"           {column += yyleng; RETURN(WORD_IMMEDIATELY);}
"CENTERFIELD"           {column += yyleng; RETURN(WORD_CENTERFIELD);}
```

[. . .]

```
"Z"                     {column += yyleng; RETURN(WORD_Z);}
"3"                     {column += yyleng; RETURN(WORD_3);}
"P"                     {column += yyleng; RETURN(WORD_P);}

{word}                  {column += yyleng;
                         fprintf(stderr,
                                 "line %d, column %d: scan error, invalid word \'%s\'\n",
                                 line, column, yytext
                         );
                         exit(1);
                        }
{whtspc}                {column++;}
<<EOF>>                 {column++; RETURN(EOI); /* End Of Input */}
[^ \t\n]                {column += yyleng;
                         fprintf(stderr,
                                 "line %d, column %d: scan error, invalid token \'%s\'\n",
                                 line, column, yytext
                         );
                         exit(1);
                        }
\n                      {line++; column = 0;}

%%

int yywrap() {return 1;}

/* main() {while(1) yylex();} */
```

B.6 An extract of atcparse.y

```
%{
#define YYERROR_VERBOSE 1
#include "atclib.h"

extern yytext[];

char* parsedLine;
%}

%token          EOI
%token          WORD_0 WORD_1 WORD_2 WORD_3 WORD_4 WORD_5 WORD_6 WORD_7 WORD_8 WORD_9 WORD_A
WORD_ABLE WORD_ABOVE WORD_ACCELERATE WORD_ADDITIONAL WORD_ADVISE WORD_ADVISORIES WORD_AERO
```

[. . .]

```
WORD_Y WORD_YANKEE WORD_YOU WORD_YOUR WORD_Z WORD_ZERO WORD_ZULU

%start        start

%%

start :
          atc_block start     {}
        | EOI                 {printf("\n=> Input accepted.\n"); exit(0);}
        ;


nonterminal_0 :
          WORD_0
        | WORD_ZERO
        ;

nonterminal_1 :
          WORD_1
        | WORD_ONE
        ;

nonterminal_2 :
          WORD_2
        | WORD_TWO
        ;

nonterminal_3 :
          WORD_3
        | WORD_THREE
        ;

nonterminal_4 :
          WORD_4
        | WORD_FOUR
        ;

nonterminal_5 :
          WORD_5
        | WORD_FIVE
        ;
```

```
nonterminal_6 :
          WORD_6
        | WORD_SIX
        ;

nonterminal_7 :
          WORD_7
        | WORD_SEVEN
        ;

nonterminal_8 :
          WORD_8
        | WORD_EIGHT
        ;

nonterminal_9 :
          WORD_9
        | WORD_NINE
        | WORD_NINER
        | WORD_NINA
        ;

nonterminal_10 :
          WORD_1 WORD_0
        | WORD_TEN
        ;

nonterminal_11 :
          WORD_1 WORD_1
        | WORD_ELEVEN
        ;

nonterminal_12 :
          WORD_1 WORD_2
        | WORD_TWELVE
        ;

point :
          '.'
        | WORD_POINT
        ;

pos_digit :
```

```
          nonterminal_1
        | nonterminal_2
        | nonterminal_3
        | nonterminal_4
        | nonterminal_5
        | nonterminal_6
        | nonterminal_7
        | nonterminal_8
        | nonterminal_9
        ;

digit :
          nonterminal_0
        | pos_digit
        ;

digit_seq :
          digit_seq_rec1
        ;

digit_seq_rec1 :
          digit
        | digit_seq_rec1 digit
        ;

number :
          nonterminal_0
        | pos_digit number_opt1
        ;

number_opt1 :
          digit_seq
        | /* empty */
        ;

number_gt_one :
          nonterminal_2
        | nonterminal_3
        | nonterminal_4
        | nonterminal_5
        | nonterminal_6
        | nonterminal_7
        | nonterminal_8
```

```
        | nonterminal_9
        | pos_digit digit_seq
        ;
```

[. . .]

```
block02 :
          sentence03
                {printf("\n ż Instance of BLOCK-rule <block02> accepted.");}
        ;

sentence03 :
          WORD_CONTACT sentence03_grp1 ffunction sentence03_opt1 sentence03_opt2 '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence03> accepted.");}
        ;

sentence03_grp1 :
          fname
        | lname
        ;

sentence03_grp2 :
          time
        | fix
        | altitude
        ;

sentence03_opt1 :
          frequency
        | /* empty */
        ;

sentence03_opt2 :
          WORD_AT sentence03_grp2
        | /* empty */
        ;

block03 :
          sentence04
                {printf("\n ż Instance of BLOCK-rule <block03> accepted.");}
        ;

sentence04 :
          WORD_CHANGE WORD_TO WORD_MY WORD_FREQUENCY frequency '.'
```

```
                {printf("\n ż Instance of SENTENCE-rule <sentence04> accepted.");}
        ;

block04 :
          sentence05
                {printf("\n ż Instance of BLOCK-rule <block04> accepted.");}
        ;

sentence05 :
          WORD_REMAIN WORD_THIS WORD_FREQUENCY '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence05> accepted.");}
        ;

block05 :
          sentence06
                {printf("\n ż Instance of BLOCK-rule <block05> accepted.");}
        ;

sentence06 :
          WORD_TRAFFIC ',' sentence06_grp2 '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence06> accepted.");}
        ;

sentence06_grp1 :
          altitude
        | WORD_ALTITUDE WORD_UNKNOWN
        ;

sentence06_grp2 :
          clock_az
        | direction ',' miles ',' sentence06_opt1 rel_movement ',' sentence06_opt2 sentence06_grp1
        ;

sentence06_opt1 :
          quad WORD_BOUND ','
        | /* empty */
        ;

sentence06_opt2 :
          craft_type ','
        | /* empty */
        ;
```

```
block06 :
          block06a
                {printf("\n ż Instance of BLOCK-rule <block06> accepted.");}
        | block06b
                {printf("\n ż Instance of BLOCK-rule <block06> accepted.");}
        ;

block06a :
          sentence07
                {printf("\n ż Instance of BLOCK-rule <block06a> accepted.");}
        ;

sentence07 :
          WORD_TRAFFIC ',' sentence07_grp1 direction WORD_OF sentence07_grp2 ',' direction WORD_BOUND ',' sentence07_opt1 sentence07_grp3 '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence07> accepted.");}
        ;

sentence07_grp1 :
          miles
        | minutes WORD_MINUTES
        ;

sentence07_grp2 :
          fname
        | fix
        ;

sentence07_grp3 :
          altitude
        | WORD_ALTITUDE WORD_UNKNOWN
        ;

sentence07_opt1 :
          craft_type ','
        | /* empty */
        ;

block06b :
          sentence08
                {printf("\n ż Instance of BLOCK-rule <block06b> accepted.");}
        ;

sentence08 :
```

```
          WORD_TRAFFIC ',' WORD_NUMEROUS WORD_TARGETS WORD_VICINITY sentence08_grp1 '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence08> accepted.");}
        ;

sentence08_grp1 :
          fname
        | fix
        ;
```

[. . .]

```
block51 :
          sentence85
                {printf("\n ż Instance of BLOCK-rule <block51> accepted.");}
        ;

sentence85 :
          WORD_MAINTAIN sentence85_grp1 sentence85_opt1 '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence85> accepted.");}
        ;

sentence85_grp1 :
          altitude
        | block_altitude
        ;

sentence85_opt1 :
          WORD_AT speed
        | /* empty */
        ;

block52 :
          sentence86
                {printf("\n ż Instance of BLOCK-rule <block52> accepted.");}
        ;

sentence86 :
          WORD_RESUME WORD_NORMAL WORD_SPEED '.'
                {printf("\n ż Instance of SENTENCE-rule <sentence86> accepted.");}
        ;

%%

int yyerror(char *s) {
```

```
        fprintf(stderr, "line %d, column %d: %s\n", line, column, s);
        return 0;
}

void initVariables() {
  line = 1;
  column = 0;
}

int main(void) {
        initVariables();
        yyparse();
        return 0;
}
```

B.7 An extract of `atc.tab.h`

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define EOI              257
#define WORD_0           258
#define WORD_1           259
#define WORD_2           260
#define WORD_3           261
#define WORD_4           262
#define WORD_5           263
#define WORD_6           264
#define WORD_7           265
#define WORD_8           266
#define WORD_9           267
#define WORD_A           268
#define WORD_ABLE        269
#define WORD_ABOVE       270
#define WORD_ACCELERATE  271
#define WORD_ADDITIONAL  272
#define WORD_ADVISE      273
#define WORD_ADVISORIES  274
#define WORD_AERO        275
```

[. . .]

```
#define WORD_Y           648
#define WORD_YANKEE      649
```

```
#define WORD_YOU          650
#define WORD_YOUR         651
#define WORD_Z            652
#define WORD_ZERO         653
#define WORD_ZULU         654

extern YYSTYPE yylval;
```