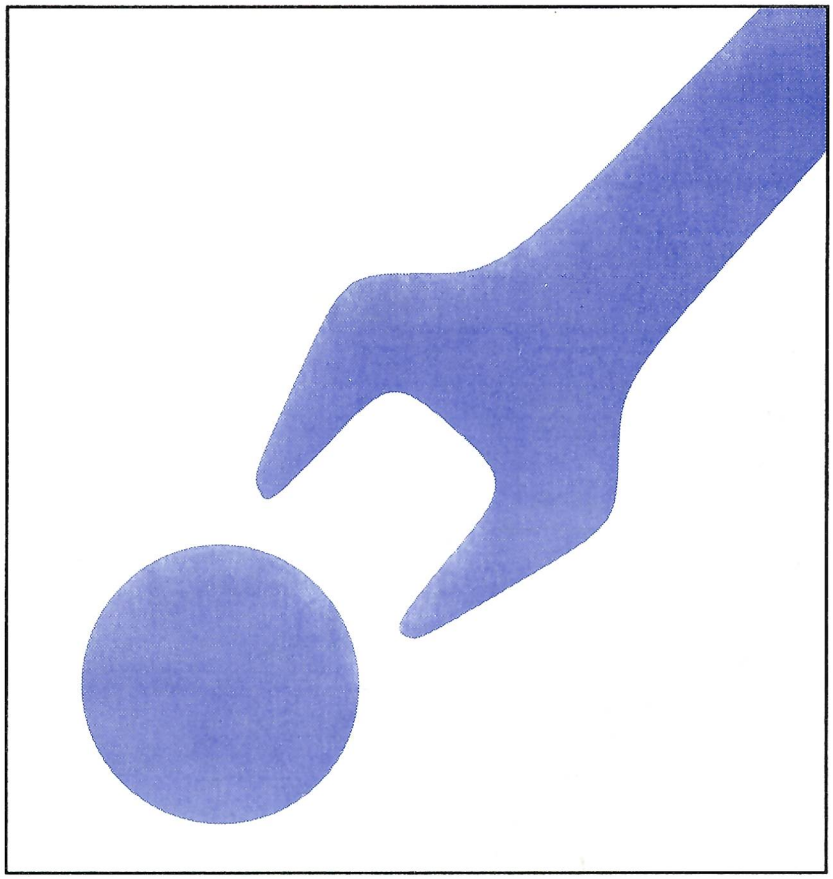
ToolMaker



Reference Manual

Version 2.0

ToolMaker is a software product from SoftLab. The ToolMaker product contains proprietary information not to be disclosed to third party. This applies to this manual, the library files and any software delivered from SoftLab as a part of the ToolMaker package. Refer to your license agreement for details.

Licenses for ToolMaker may be obtained from:


```
SoftLab
Datalinjen 1
S-583 30 LINKÖPING
SWEDEN

Phone:    Nat. 013 - 21 24 70
          Int. +46 13 21 24 70
```


Suggestions and error reports should also be forwarded by your local ToolMaker contact person to the above address.

Electronic mail contacts (limited service only unless otherwise explicitly stated) may be sent to:


```
tools@softlab.se
```


This version of the manual was printed on April 12, 1994.

Part I

ToolMaker
System Description

1 INTRODUCTION 21

2 CONCEPTS AND ASSUMPTIONS 22

3 SYSTEM DESCRIPTION 23

3.1 The ToolMaker Components 23

3.2 Structure of a ToolMaker Component 25

4 A TOOLMAKER BASED SYSTEM 27

4.1 The ToolMaker Description File 27

4.2 ParserMaker 27

4.3 ScannerMaker 28

4.4 ListerMaker 28

4.5 Putting It All Together 28

5 THE TOOLMAKER DESCRIPTION FILE 30

5.1 The Options Section 30

5.2 The Import Section 34

5.3 The Srcp Section 34

5.4 The Token Section 36

6 USING A TOOLMAKER BASED SYSTEM 38

6.1 Import 38

6.2 Export 38

6.3 Compiling and Linking 38

6.4 Run-time 39

7 COMMON DEFINITIONS 40

7.1 Description File Option Format 40
7.2 Command Line Option Format 40
7.3 Option Precedence 42
7.4 Prefix Management 42
7.5 Description File Sections 43
7.6 Escape Character 43
7.7 Strings 44

A Appendix: THE PL/0 EXAMPLE 45

B Appendix: ERROR MESSAGES 49

C Appendix: SYNTAX NOTATION 52

D Appendix: REFERENCES 53

Part II

ParserMaker Reference Manual

1 INTRODUCTION 59

2 CONCEPTS AND ASSUMPTIONS 60

2.1 Backus-Naur Form (BNF) 60
2.2 Extended Backus-Naur Form (EBNF) 60
2.3 Productions 62
2.4 Semantic Actions 63
2.5 Grammar Attributes 64
2.6 Grammar Ambiguity and LALR-conflicts 68
2.7 Error Recovery Principles 72

3 PARSER PRODUCTION 76

3.1 Creating a Running Parser 76
3.2 The ParserMaker Description File 79
3.3 The ToolMaker Common Description File 100
3.4 The List File 100
3.5 The Vocabulary File 102

4 THE PARSERMAKER COMMAND 104

4.1 Parameters 104
4.2 Options 104
4.3 Example 105

5 PARSER RUN-TIME USAGE 106

5.1 Principles of Operation 106

5.2 Run-Time Interface 107

A Appendix: THE PL/0 EXAMPLE 111

B Appendix: ERROR MESSAGES 117

C Appendix: DESCRIPTION LANGUAGE 123

D Appendix: TARGET LANGUAGE DETAILS .. 128

Part III

ScannerMaker Reference Manual

1 INTRODUCTION 133

2 CONCEPTS AND ASSUMPTIONS 134

2.1 ScannerMaker and Scanner 134
2.2 Vocabulary 134
2.3 Token 134
2.4 Scanner 134
2.5 Scanner and Inheritance 134
2.6 Screening 134
2.7 Token and Source Position Definitions 135

3 SCANNER PRODUCTION 136

3.1 Creating a Running Scanner 136
3.2 The ScannerMaker Description File 138
3.3 The ToolMaker Common Description file 160
3.4 The Vocabulary file 160

4 THE SCANNERMAKER COMMAND 162

4.1 Parameters 162
4.2 Options 162

5 SCANNER RUN-TIME USAGE 164

5.1 Principles of Operation 164
5.2 Run Time Interface 167
5.3 Recursive Calls and Continued Scanning 170

A Appendix: THE PL/0 EXAMPLE 172

B Appendix: ERROR MESSAGES 176

C Appendix: DESCRIPTION LANGUAGE 182

D Appendix: TARGET LANGUAGE DETAILS .. 188

Part IV

ListerMaker Reference Manual

1 INTRODUCTION 197

2 CONCEPTS AND ASSUMPTIONS 198

2.1 ListerMaker and Lister 198
2.2 Source Positions 198
2.3 Messages 198
2.4 Error Message Templates and Insert Strings 199
2.5 Severities 199
2.6 Listing Types 200

3 LISTER PRODUCTION 201

3.1 The ListerMaker Description File 201
3.2 The ToolMaker Common Description File 204

4 THE LISTERMAKER COMMAND 206

4.1 Parameters 206
4.2 Options 206

5 LISTER RUN-TIME USAGE 208

5.1 Principles of Operation 208
5.2 Run Time Interface 211
5.3 Messages Templates 218

A Appendix: THE PL/0 EXAMPLE 220

B Appendix: ERROR MESSAGES 221

C Appendix: TARGET LANGUAGE DETAILS .. 222

Part V

Toolmake
Reference Manual

1 INTRODUCTION 231

2 PRINCIPLES OF OPERATION 232

3 THE TOOLMAKE COMMAND 233

3.1 Parameters 233

3.2 Options 233

4 TOOLMAKE RUN-TIME USAGE 234

4.1 Subsystem Name 234

4.2 ToolMake Commands 234

A Appendix: ERROR MESSAGES 237

B Appendix: FILE GENERATION DETAILS 240

Part VI

Index

Part I

ToolMaker
System Description

6.4 Run-time 39

7 COMMON DEFINITIONS 40

7.1 Description File Option Format 40
7.2 Command Line Option Format 40
7.3 Option Precedence 42
7.4 Prefix Management 42
7.5 Description File Sections 43
7.6 Escape Character 43
7.7 Strings 44

A Appendix: THE PL/0 EXAMPLE 45

A.1 Directory Listing 45
A.2 pl0.tmk - the ToolMaker Description File 45
A.3 pl0.c - the Main Program 46

B Appendix: ERROR MESSAGES 49

B.1 Message Format 49
B.2 Messages Explanations 50
B.3 License Errors 51

C Appendix: SYNTAX NOTATION 52

D Appendix: REFERENCES 53

1 INTRODUCTION

ToolMaker is a complete software package for creating software tools. ToolMaker is specially intended for creating analysing tools like compiler front-ends, pretty-printers or translators but is equally well suited for simplifying the creation of text or data conversion tools.

ToolMaker is built on sound and well-tried software technology, such as parsing techniques, various data compression algorithms and modularisation, all to make ToolMaker a fast, flexible and reliable software product.

Although the utmost has been made to simplify the usage of ToolMaker it still requires some knowledge about parsing theory, the notion of grammars and regular expressions. Understanding the method of software generation techniques is also most helpful, as this is the basis for the complete ToolMaker package.

A novice user should carefully study the introductory chapters in all the parts of the ToolMaker documentation for the various tools and the example in this part before attempting to build his own tool using ToolMaker.

The more experienced user, familiar with similar products (notably YACC and LEX), should at least read chapter 4, *A TOOLMAKER BASED SYSTEM* on page 27 in this part of the document and then refer to the reference chapters for each individual Maker in the ToolMaker kit.

2 CONCEPTS AND ASSUMPTIONS

ToolMaker is a set of tools to generate the input analysing part of a complete tool, a ToolMaker-based *subsystem.* A subsystem is denoted by a name - a *system name*. This system name may be prepended to many of the function and type names exported by a ToolMaker based subsystem, a *system prefix*.

The three main parts of a ToolMaker-based subsystem (the *scanner*, *parser* and *lister*) may all be generated by the appropriate *Maker*, ParserMaker, ScannerMaker or ListerMaker. Each Maker is controlled by a *description file* in which information particular to the Maker is placed and a *ToolMaker common description file* containing common information and options.

To create a complete tool, add your analysing modules to the ToolMaker generated modules and link the complete system.

Throughout this document the term *Maker* will be used to refer to any of the different software tools included in the ToolMaker package.

ToolMaker generated parsers, scanners and listers are automatically interfaced to each other using common declarations and usages of *source positions*, *token codes* and *error messages*.

A *source position* is a data structure describing a position in the input, this could for example point to the beginning of the 'BEGIN' as the first token in a Pascal block. This data structure may contain line, column and file information and is automatically collected by the generated scanner thus giving the implementor efficient handles for indicating errors and other information in relation to the analysed input. Source positions could for example be stored as attributes in the decorated abstract syntax tree of the input.

The scanner combines the input characters into *tokens* which also are represented in a data structure returned to the parser. The information about which token was recognized is represented by a value in the token data structure, the *token code*. The token structure may also contain additional user defined attributes such as integer or string values or the identifier name.

Errors often occur in the input and a ToolMaker based system is well prepared to handle, recover from and report the errors to the user. The generated parser calls an error handler interface to report the syntax errors it encounters and recovers from. The generated error handler module by default prepares this information into message logging calls to the Lister. Such a call carries source position information, a reference to a message template and optionally additional information to be inserted into the message to form an actual *error message* which can be shown in listings or retrieved through procedure calls. The logging of messages is also available as functions to be used by the implementor for example in later, e.g semantic, passes of the analysis.

3 SYSTEM DESCRIPTION

3.1 The ToolMaker Components

The ToolMaker kit consists of the following highly integrated components:

- *ParserMaker* - an efficient LALR parser generator with automatic error recovery and much more.
- *ScannerMaker* - a scanner generator for very fast table driven lexical analysis.
- *ListerMaker* - a message handling system with high flexibility.
- *Incremental Macro Processor* - a source code processor with many powerful features (described in separate documentation).
- *Toolmake* - a support facility simplifying setting up a new ToolMaker based system.

In addition ToolMaker also comes with a suggestion for a main program showing how the various run-time modules may be connected creating a functioning tool, and a skeleton for an intelligent **makefile** (where this is available) to maintain and automatically produce up-to-date versions of your tool.

The functionality of the components are briefly described below, but for details refer to the reference document for the individual component.

3.1.1 ParserMaker

The major component in ToolMaker is the LALR-parser generator ParserMaker. This component generates parsers from grammars, i.e. from a formal textual description of a language it creates programs that checks its input according to that grammar.

The generated parser not only accepts or refuses its input but also contains various error recovery routines. This means that when an incorrect input is encountered it automatically tries to correct it so that the rest of the input also may be analysed. This automatically generated error recovery is possible to further fine-tune by giving ParserMaker additional information.

In addition to the actual grammar it is also possible to specify semantic actions for each production (or rule). A semantic action is target language code provided by the implementor that will be executed each time the parser reduces the input according to the corresponding rule, i.e. when that input is encountered. The code in the semantic actions may refer to data synthesised in other terminals or non-terminals, so called attributes. Using these attributes a representation of the input may be built for example.

3.1.2 ScannerMaker

The ScannerMaker tool produces lexical analysers from a description of regular expressions to recognise. The emphasis when designing this tool has been speed as a major part of the execution time of a software tool often is to read the input and compose it into tokens.

ScannerMaker supports multiple vocabularies (sets of tokens to recognize) and scanners making it easy to implement context sensitive scanning.

The connection between ParserMaker and ScannerMaker is almost completely seamless as ScannerMaker takes as input not only a file describing the regular expressions but also a file produced by ParserMaker containing the tokens found in the grammar.

3.1.3 ListerMaker

A *Lister* is a subroutine package to handle error messages and to produce listing files. ListerMaker is a tool to create and customize such modules also included in the ToolMaker kit. This makes it very easy to produce and handle listings and error messages.

3.1.4 Incremental Macro Processor

To be able to efficiently produce the source code to compile from the various generated files a specially optimized text processor is also included in ToolMaker. The Incremental Macro Processor allows text to be conditionally included, sections defined and inserted at other places, variables to be expanded and so on.

ParserMaker, ScannerMaker and ListerMaker all uses the facilities of the Incremental Macro Processor to automatically generate compilable source code in an efficient manner.

The Incremental Macro Processor is delivered as a separate tool with the ToolMaker package and is documented in a separate manual (the *IMP Reference Manual*).

3.1.5 Toolmake

Toolmake makes it very simple to set up a new subsystem based on ToolMaker. Using a dialogue you specify what parts you want to generate using ToolMaker, ***toolmake*** creates the appropriate files in the current directory.

You may also select the level of information already in the various files (description files, main module suggestion etc.), minimal, normal contents or a complete example (see *THE PL/0 EXAMPLE* on page 45, page 111, page 172 and page 220).

3.2 Structure of a ToolMaker Component

This section describes the general structure of a ToolMaker component, its inputs, the processing it performs and the resulting outputs. The description is in generic terms and applies to ParserMaker, ScannerMaker and ListerMaker. Any major discrepancies between the three are mentioned where appropriate.

3.2.1 Description Files

A description file is the input to a component. This file contains information for the actual generation process.

The description file for ParserMaker contains the description of the grammar, and for ScannerMaker the regular expressions to be assembled to tokens. The ListerMaker description file contains primarily the message templates.

A description file contains sections containing different information, such as options and the rules. Both ScannerMaker and ParserMaker have rules sections where the grammar and regular expressions respectively are described in a meta-language.

3.2.2 The Maker

Each actual Maker reads and analyses its description file, producing any error messages on the screen and a listing file if requested.

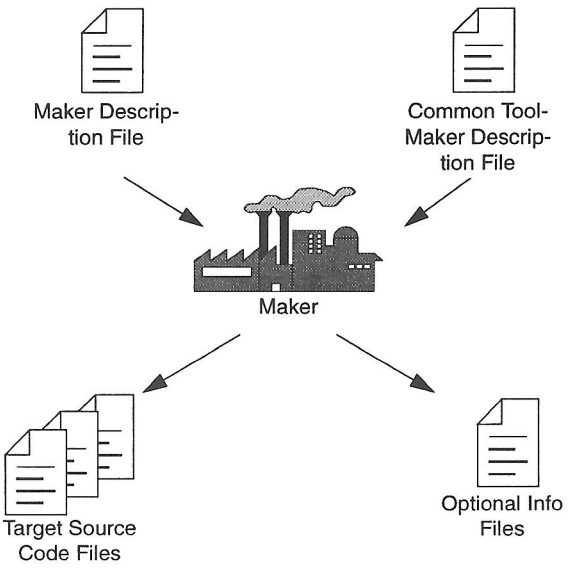


Fig. 1 File flow in a Maker.

After having analysed the Makers own description file and the optional ToolMaker Common Description file (containing definitions common for all Makers, refer to *THE TOOLMAKER DESCRIPTION FILE* on page 30) the Maker produces output in the form of table files. The table files contain data and code definitions to complement skeleton files which are provided and may be modified. As a last step the table file is processed together with the skeleton files (using the Macro processor IMP) to produce the actual source files.

> *Note:* This is a major difference with respect to version 1 of ToolMaker where the generation of source files from the tables was *not* performed automatically by the Makers.

Also a source file is never generated unless necessary, i.e. unless the contents of the file has changed. This can make enormous differences in compile times for large systems (when using automatic build tools such as ***make***) as files that are dependent on ToolMaker generated source files need not be recompiled unless a real change has been made. For small systems the gain may be small in comparison with times needed to compare the new files so the possibility to force generation (thus avoiding the comparison) is available.

3.2.3 Skeleton Files

The skeleton files are files in IMP format which when processed by the Incremental Macro Processor together with the table files generated by the Maker will produce the actual data and code which constitutes the compilable source for the parser, scanner or lister.

The skeleton files resides in a special directory delivered with ToolMaker. There is one separate subdirectory for each target language available.

Under particular circumstances it may be necessary to modify the skeleton files. The modified files should be kept separate from the standard skeleton files and pointed to by using the `library` option of the Makers.

> *Note:* To modify the skeletons thorough understanding of the mechanisms of the Maker and the Incremental Macro Processor is required.

4 A TOOLMAKER BASED SYSTEM

In this section we will go through a simple but complete example using ToolMaker. The example will briefly describe all the files needed, their relationships and the running of each maker.

This chapter shows an example with the system name **pl0**. The example is also available with the ToolMaker distribution and some of the files are shown in appendix A, *THE PL/0 EXAMPLE*, on page 45. The example uses the 'ansi-c' language as its target language.

Note that the figures below only show the files present in your directory, and thus the skeleton files are not shown, but of course they are also needed for the generation.

4.1 The ToolMaker Description File

ToolMaker requires one common description file containing information about common data declarations, such as source position and token representations, options common to all the Makers and various other definitions.

The ToolMaker description file is called **pl0.tmk**. See appendix section A.2, *pl0.tmk - the ToolMaker Description File* on page 45 for an example of the contents of a ToolMaker description file.

The processing of the ToolMaker description file is performed automatically by each of the Makers as a part of their normal processing.

4.2 ParserMaker

The next file needed is the ParserMaker description file. This file should contain the grammar of the input to your tool. In addition optional sections may be included to inform ParserMaker of your selected options (like packing strategies, target language etc.), error recovery information a.s.o. For more details refer to the *ParserMaker Reference Manual*.

This file we call **pl0.pmk** and is input to ParserMaker with the command

```
pmk pl0
```

This will produce the following output files (assuming no errors were detected):

- **pl0.voc** - generated vocabulary file containing all tokens found in the grammar and their corresponding token codes.
- **pl0.pml** - optionally generated listing file containing the input, error messages and other information.

- **pl0Parse.h**, **pl0Parse.c**, **pl0PaSema.c** 'ansi-c' source for the parser proper and semantic actions from the description file.
- **pl0Err.h**, **pl0Err.c** 'ansi-c' source for error message interfacing (optional).
- **pl0.pmt** - generated tables and source code sections in IMP format (normally removed after successful source file generation).

4.3 ScannerMaker

Using ScannerMaker is much like using ParserMaker. It needs a description file, **pl0.smk**, describing the regular expressions the generated scanner should recognise. As with the ParserMaker description file this file also may have additional sections defining options and target source code for various code hooks, but for details refer to the *ScannerMaker Reference Manual*.

In addition ScannerMaker also accepts the vocabulary file produced by ParserMaker, in this example **pl0.voc**, making the transfer of vocabulary information from the grammar to the scanner generating process fully automatic.

From this input ScannerMaker generates the target language source files, **pl0Scan.h**, **pl0Scan.c** and **pl0ScSema.c**, a table file, **pl0.smt**, which is normally removed, and, optionally, a listing file, **pl0.sml**.

4.4 ListerMaker

The major part of the ListerMaker description file (**pl0.lmk**) describes the error message templates used when producing the actual messages in the listing. The messages may be divided into sections each having the same messages in different languages for example.

ListerMaker produces the source files for the lister, **pl0List.c** and **pl0List.h**. The error message templates to be used during run-time may either be compiled into the source files as data or be placed in a separate file, in this example named **pl0.msg**, which is generated by ListerMaker from the description file.

The message templates may contain markers where to insert additional text which will be supplied during run-time.

4.5 Putting It All Together

For a complete system we also need a main program and any additional modules needed to analyse the input, handling symbol tables, code generation or what ever is needed to complete your tool.

To make the ToolMaker generated parts function the main program, **pl0.c**, must initialise the lister and the scanner and then call the parser. As a last action a call to the lister to produce a listing file or show the error messages on

the screen is advisable. In appendix section A.3, *pl0.c - the Main Program* on page 46 an example of a main program can be found.

The files in this example are all automatically produced using the ***toolmake*** utility (see *Toolmake Reference Manual*) if the contents level *example* is used. To study their contents, create a new directory and set up a ToolMaker environment with level *example* as described in the *Toolmake Reference Manual*. Using ***toolmake*** creates all necessary files, including a main program suggestion and a **makefile** (or command file depending on your environment) to keep track of dependencies between description files, table files, source code, object code and executable. The **makefile** must be updated to include any additional modules that are to be included in the complete tool.

5 THE TOOLMAKER DESCRIPTION FILE

The ToolMaker description file is a file which describes options and data structures common to all the Makers in the ToolMaker package. The file is necessary whenever more than one Maker is utilized. However when one single Maker is used the information normally kept in the ToolMaker description file may be specified in the description file of that Maker. This makes it possible to use all tools in the ToolMaker package without duplicating information while still making it easy to use a single tool without distributing information across many input files.

The preferred file extension for the ToolMaker description file is **.tmk**, and the syntax of the ToolMaker description file is:

```
<description file> ::=
       [ <options section> ]
       { <import section> | <token section>
          | <srcp section> }
```

For a brief description of the notation used to describe the syntax of the description files refer to appendix C, *SYNTAX NOTATION*, on page 52.

5.1 The Options Section

```
<options section> ::=
     '%%OPTIONS' <directive> {<directive>} ['%%END']

<directive> ::=
       <verbose directive>
     | <target directive>
     | <os directive>
     | <prefix directive>
     | <library directive>
     | <escape directive>
     | <width directive>
     | <height directive>
     | <generate directive>
     | <force directive>
```

The *options section* of the ToolMaker description file can be used to set options common to more than one tool. Corresponding options and options special for each Maker can be given as command line switches to each tool at its invocation. Options given in other description files override those given in the ToolMaker description file.

> *Note:* Directives given in the ToolMaker description file always apply when generating the common declarations for the ToolMaker based components (in 'c' the **tmCommon.h**) even if that option is set to another value locally for a particular Maker (for example if the target directive is set in the description file for a Maker the common declarations will still be generated in the target language defined in the ToolMaker description file).

The directives can be given in any order. If a directive is repeated, only the last one given is used.

> *Note:* If a directive is explicitly stated in the *options section*, all its default option values are turned off. This is especially important for set valued directives and means that all desired options must be included when a directive is stated, not just the desired extra ones.

5.1.1 The Verbose Directive

```
<verbose directive> ::=
     ['NO'] 'VERBOSE' ';'
```

The verbose directive controls the amount of information that is output on the terminal during the generation processes.

If `NO VERBOSE` is specified, the generation process will be *silent*, and only error messages will be output to the terminal.

If `VERBOSE` is specified, the *progress* of the generation process will be logged to the terminal, and *timing figures* for the various phases will be supplied.

The default is:

```
No Verbose;
```

5.1.2 The Target Directive

```
<target directive> ::=
     'TARGET' <quoted string> ';'
```

This option specifies for which language the files should be generated. The following values of the target option are supported:

`'c'`	generate files in 'c' (K&R)
`'ansi-c'`	generate files in 'c' (ANSI)
`'c++'`	generate files in 'c++'

> *Note:* The option values are case sensitive and refer to subdirectory names in the standard installation. To use the default library files the target language must be given in lower case letters.

The default setting is:

```
Target 'ansi-c';
```

> *Note:* Additional target languages may be supported. Refer to the Release Notes for the latest version.

5.1.3 The OS Directive

```
<os directive> ::=
     'OS' <quoted string> ';'
```

This option specifies for which operating system the files should be generated. The following options are supported:

`'SunOS'` generate files for Sun OS

The default setting is:

```
Os 'SunOS';
```

Note: Additional operating systems may be supported. Refer to the Release Notes for the latest version.

5.1.4 The Prefix Directive

```
<prefix directive> ::=
     'PREFIX' <quoted string> ';'
```

The prefix is used as a prefix for all *externally visible functions and types*. This prefix set (implicitly or explicitly) in the ToolMaker Common Description file is known as the system prefix and may be overridden locally for each Maker. System wide types and functions will still use the system prefix.

The default value is:

```
Prefix 'tm';
```

5.1.5 The Library Directive

```
<library directive> ::=
     'LIBRARY' <quoted string> ';'
```

The library directive instructs the ToolMaker tools to look for template files in a directory other than the default, which is **$TMHOME/lib/*target*/**, for example:

```
Library '/usr/local/lib/ToolMaker/lib/ansi-c';
```

5.1.6 The Escape Directive

```
<escape directive> ::=
       'ESCAPE' <escape character> ';'
     | 'NO' 'ESCAPE' ';'

<escape character> ::= <quoted string>
```

The escape directive allows the implementor to change the escape character in target language sections. The quoted string used to describe the escape

character must contain exactly one character which is directly interpreted as the escape character.

If NO ESCAPE is specified no character will be used as escape character in semantic target language sections.

Default:

```
Escape '`';
```

5.1.7 The Width Directive

```
<width directive> ::=
        'NO' 'WIDTH' ';'
      | 'WIDTH' <number> ';'
```

This option specifies the maximum number of characters on a line in the list file (if requested) before a line is broken into two. If NO WIDTH is specified, lines will be printed in full length.

The default value is:

```
Width 78;
```

5.1.8 The Height Directive

```
<height directive> ::=
        'NO' 'HEIGHT' ';'
      | 'HEIGHT' <number> ';'
```

This option specifies the number of lines that fit on a page in the list file. If NO HEIGHT is specified, the list file will not be divided into pages.

The default value is:

```
Height 60;
```

5.1.9 The Generate Directive

```
<generate directive> ::=
     'GENERATE' <generate> { ',' <generate> } ';'

<generate> ::=
     'TABLES' | 'SOURCE'
```

The generate directive instructs the Makers on which output to generate. The intermediate table files are normally not kept after successfully generating the target language source, but by using this option any combination of tables and/or source files may be generated.

The default for all Makers is to generate only source files. To generate tables this option must be used.

Note: Setting `Generate tables;` will *only* generate the table file. To get both tables and source use `Generate tables, source;`.

5.1.10 The Force Directive

```
<force directive> ::=
          [ 'NO' ] 'FORCE' ';'
```

The force directive instructs the Makers to force the generation of source files, even though it might not be necessary, thus overriding the built-in functionality to not overwrite any source files that would contain identical information.

The default value is naturally:

```
No Force;
```

5.2 The Import Section

```
<Import section> ::=
     '%%IMPORT' <target language code> ['%%END']
```

Any definition needed throughout the generated subsystem should be imported by placing appropriate declarations or includes in the *import section* in the ToolMaker Common Description file. An example are types referenced within the token and srcp sections. The section should contain *declarations in the target language*. The text representing the target language code, is copied unformatted to the output file.

5.3 The Srcp Section

```
<srcp section> ::=
     '%%SRCP'
     [<srcp name>]
     [{<srcp member>}]
     ['%%END']

<srcp name> ::=
     'NAME' <identifier> ';'

<srcp member> ::=
     <srcp kind> <identifier> <opt declaration> ';'

<srcp kind> ::=
     'LINE' | 'COLUMN' | 'POSITION' | 'FILE'

<opt declaration> ::=
     [ '%%' <target language code> '%%' ]
```

The srcp section defines the system source position type. This definition is for example used by the generated parser in its scanner and error handling interfaces. It includes name for the srcp (source position) type and the srcp members.

Language dependent declarations, i.e. descriptions of the implementation, are defined in the optional declaration part of srcp members. These declarations must either be used throughout the section or skipped entirely. If the language dependent declarations are used, ToolMaker will generate a type declaration. If they are not used, an external type definition must be imported (in the import section). Refer also to the section below on the %%TOKEN section.

The four kinds of srcp members are *file, line, column* and *position*. If a member is stated in the srcp section, code will be generated by ToolMaker to handle it. A ScannerMaker generated scanner automatically deduces which type of source position information to calculate from the fields present in the *srcp section*.

To be able to access the line component of a source position variable, the `LINE` member should be used. The specified identifier will be used as field name of the line component of a source position variable. This information indicates on which line this token started (assuming a normal text file as input).

To be able to access the column component of a source position variable, the `COLUMN` member should be used. The specified identifier will be used as the field name of that component. The identifier is the field name of the column component of a source position variable. This field will indicate at which column on the line the token started, the first position on a line being column 1 (one).

To be able to access the file component of a source position variable, the `FILE` member should be used. The specified identifier will be used as field name of the file component of a source position variable. This numbering of input sources is *not* automatically performed by a ScannerMaker generated scanner since this information must be coordinated between user defined code in the scanner that opens a new stream of input and other parts of the user code (refer to *Principles of Operation* in the *ListerMaker Reference Manual*, page 208 for hints on how to use and initialise the file component).

Using a ListerMaker generated lister module requires the use of the *line* and *column* fields. If the `Include;` or `Listings multiple;` options of ListerMaker are used the *file* component must also exist.

To be able to access the position component of a source position variable, the `POSITION` member should be used. The specified identifier will be used as the field name of that component.

Example:

```
%%SRCP
     NAME pmSrcp

     LINE   lin %% int lin %%;
     COLUMN col %% int col %%;
```

Instead of repeating the name of the identifier in the target language declaration, it is possible to use a '%1' as a placeholder for the name. The placeholder will automatically be substituted with the name in the produced code.

Example:

```
%%SRCP
     NAME pmSrcp;

     LINE   lin %% int %1 %%;
     COLUMN col %% int %1 %%;
```

Note: The opening %% must be followed by a space.

5.4 The Token Section

```
<token section> ::=
     '%%TOKEN'
     [<token name>]
     [<token code>]
     [<token srcp>]
     [<token attributes>]
     ['%%END']

<token name> ::=
     'NAME' <identifier> ';'

<token code> ::=
     'CODE' <identifier> <opt declaration> ';'

<token srcp> ::=
     'SRCP' <identifier> <opt declaration> ';'

<token attributes> ::=
     'ATTRIBUTES'
          <token attribute> {',' <token attribute>} ';'

<token attribute> ::=
     <identifier> <opt declaration>

<opt declaration> ::=
     [ '%%' <target language code> '%%' ]
```

The token section defines the system token type. This definition is used by the generated parser in its error handling interfaces, and is also the type returned by the generated scanner. The definition includes the name for the token type, the token code member, the token source position (`srcp`) member and the token attribute members.

Language dependent declarations, i.e. a description of the implementation, are defined in the optional declaration part of code, srcp and attributes. These declarations must either be used throughout the section or skipped entirely. If the language dependent declarations are used, ToolMaker will generate a type declaration in the target language. If they are not used, an external type definition must be imported (in the import section) and the token section only defines the names of the fields. Any target language dependent terminators (';' for example) should *not* be included in the target language code representing the declaration (within the %%'s). These will be provided by the generation process. Any special tokens *inside* the declaration, such as ':' in Pascal declarations should be included.

Example:

```
%%TOKEN
     CODE code %% int code %%;
     ATTRIBUTES
          ival %% int ival %%,
          sval %% String sval %%;
```

Instead of repeating the name of the identifier in the target language declaration, it is possible to use '%1' as a placeholder for the name. The placeholder will automatically be substituted with the name in the produced code. For example:

```
%%TOKEN
     CODE code %% int %1 %%;
     ATTRIBUTES
          ival %% int %1 %%,
          sval %% String %1 %%;
```

Note: The opening %% must be followed by a space.

An example of the use of an external imported token type which will not generate a type declaration but use an imported one (the names of the fields must be made known to the Makers anyway):

```
%%IMPORT
#include "Token.h"

%%TOKEN
     NAME Token;
     CODE code;
     SRCP sourcePos;
     ATTRIBUTES
          ival;
          sval;
```

6 USING A TOOLMAKER BASED SYSTEM

6.1 Import

A ToolMaker based system may be viewed as having the ToolMaker generated parts as an autonomous subsystem of the complete application. Any types and functions needed within the user specified code of any of the generated components can be imported to the complete subsystem by placing relevant import-statements in the %%IMPORT-section of the **.tmk** file. These imports will be valid through out the ToolMaker-based subsystem. In addition the %%IMPORT sections in the various description files may be utilized to import specifications to that part of the subsystem only.

6.2 Export

User-defined functions that should be exported from the various parts in the ToolMaker based subsystem to the rest of the system (application) may be defined by placing appropriate export statements in the %%EXPORT-section of the description file. The export declarations will then be included in the generated header or definition files for that module (e.g. **Parse.h** if the declaration was placed in the **.pmk** file).

6.3 Compiling and Linking

To compile a complete ToolMaker based system all utilized Makers must be run and the generated source files compiled before linking the application. On UNIX and UNIX-like environments a **make**-like utility is normally available to make this process automatic. The following lines outlines the use of such a utility:

```
all: tm ttParse.o ttPaSema.o ttList.o ttScan.o ttScSema.o
ttErr.o

tm: .smkstamp .pmkstamp .lmkstamp

.lmkstamp : tt.lmk tt.tmk
     lmk tt
     touch .lmkstamp

.pmkstamp : tt.pmk tt.tmk
     pmk tt
     touch .pmkstamp

.smkstamp : tt.smk tt.tmk tt.voc
     smk tt
     touch .smkstamp

ttParse.h ttParse.c ttPaSema.c ttErr.c tt.voc:
     touch tt.pmk
     make .pmkstamp

ttScan.h ttScan.c ttScSema.c:
     touch tt.smk
```

```
        make .smkstamp

ttList.h ttList.c:
        touch tt.lmk
        make .lmkstamp

#-- Dependencies between generated files
ttErr.o: ttErr.c ttCommon.h ttList.h ttScan.h
ttParse.o: ttParse.c ttCommon.h ttParse.h ttScan.h
ttPaSema.o: ttPaSema.c ttCommon.h ttScan.h
ttScan.o: ttScan.c ttCommon.h ttScan.h
ttScSema.o: ttScSema.c ttCommon.h ttScan.h
ttList.o: ttList.c ttCommon.h ttList.h
```

The `all` target indicates that the application is dependent on `tm` (which is a subtarget to create the generated sources) and on the object files acquired by compiling them. Inserting the `tm` target first will force a generation of the source files before compiling anything. The `tm` target is dependent on the hidden stamp files (**.smkstamp**, **.pmkstamp** and **.lmkstamp**) which always carry the date of the last run by the corresponding Maker. This is necessary as the Makers have the intelligent source generation method mentioned in *The Maker* on page 25, otherwise the Makers would be run again and again in an attempt to update the output files. The `touch` commands updates the marker files as soon as a Maker have been run.

The targets for the various generated source files ensures that if the files should not exist the appropriate makers are run and finally the dependencies between the generated **.c** and **.h** files are listed.

To adapt this in a complete environment add any additional object files to the list and add the appropriate dependencies.

6.4 Run-time

To call the ToolMaker generated functions at run-time the appropriate import declarations should be made (e.g. in 'c' the `#include` statements inserted). The main program should import the **Parse**, **Scan** and **List** declarations to be able to initialise the scanner, listing system and then call the parser. Refer to *ParserMaker Reference Manual*, *ScannerMaker Reference Manual* and *ListerMaker Reference Manual* for details on how this is performed. The **Common** header file may also be included in various modules of the application (e.g. to include the source position structure in external type declarations such as an abstract syntax tree).

7 COMMON DEFINITIONS

7.1 Description File Option Format

Each option is described in detail in the corresponding sections of this manual but the following types of options exist:

- boolean
- value (string or numeric)
- set

In the description files options occur in the Options section. Boolean options are specified simply by stating the name of the option, or to disable that option precede it with the word 'No'. For example

```
Errorhandler;
No Trace;
```

A value option should be followed by the value that the option should have, for example

```
Stacklimit 64;
Library '/usr/local/lib/ToolMaker';
```

Some value options also accepts a leading 'No' indicating that this option should have no value, as opposed to the default value.

```
No Prefix;
```

indicates that no prefix is to be used instead of the default value '`tm`'.

Set options take a number of values from a predefined set. For example

```
Pack GCS, Error;
```

will select the two packings from the set of available packing schemes. Again some set options accept the 'No' prefix, like

```
No Pack;
```

7.2 Command Line Option Format

All Makers in the ToolMaker kit uses the same format for command line options. Every option available in a Maker description file is also available on the command line. The exact set of command line options (and arguments) available may be listed using the special command line option **help**, for example:

```
pmk -help
```

will give the following printout:

```
Usage: pmk [-help] [options] <input file>
Options:
  -[-]verbose       enable[disable] verbose mode
  -target <lang>    generate file for target language
                          <lang>
  -os <os>          generate source files for target
                          operating system <os>
  -[-]prefix [<prefix>] set [no] parser prefix
  -library <lib>    use directory <lib> for library files
  -[-]escape [<c>] set [no] escape character
  -[-]width [<n>]  set [no] listing width
  -[-]height [<n>] set [no] listing height
  -[-]generate      select [no] generated output
                          {tables | source}
  -[-]force         do [not] force generating of source
                       code
  -[-]listerprefix [<prefix>] set [no] lister prefix
  -[-]errorhandler  enable [disable] generation of error
                       handler
  -[-]trace         enable [disable] trace mode
  -lookaheadmax <n>set max lookahead to <n>
  -shiftcost <n>    set shift cost for terminals to <n>
  -stacklimit <n>   set parse stack limit to <n> entries
  -[-]pack          set [no] table packing { row | column
                                | rds | gcs | les }
  -[-]actionpack    set [no] packing of action tables
  -[-]gotopack      set [no] packing of goto tables
  -[-]list          set [no] listings {input|grammar|
                       items|tables|statistics|info}
  -[-]optimize      set [no] optimize mode
  -[-]recovery      set [no] recovery mode { single
                                | multiple | panic }
  -[-]resolve       set [no] resolve mode
  -voc <file>       write vocabulary to <file>
  -pml <file>       write lists to <file> (if any)
  -pmt <file>       write tables to <file> (if any)
  -tmk <file>       read common options from <file>
  -help             this help information
```

This quick help is always available for all ToolMaker commands.

A boolean option may be set to true from the command line by giving its name preceded by a dash, or to false by preceding it with double dashes.

```
pmk --force -verbose ...
```

means set the 'force' flag to false and the 'verbose' flag to true.

A value option is given its value after the option name separated by a space. For example

```
pmk -stacklimit 64 ...

pmk -library ../lib ...
```

A set valued option may be given a set of values by repeating the option with different values, it will only accept one value for each occurrence. The option is initialised to the empty set when it first occurs thus completely overriding any default setting. A set valued option may be given the value of the empty set by giving the option name preceded by double dashes without any value.

```
pmk -generate tables -generate source --recovery ...
```

The command above will generate both tables and source and set the recovery option to the empty set (equivalent to 'No Recovery;')

All option names may be abbreviated on the command line as long as they are unambiguous. If the name of an option directive completely matches while at the same time being a prefix of another directive this will *not* be considered an ambiguity, the complete match will be accepted.

7.3 Option Precedence

Values of the various options in the Makers may be specified on four levels:

- on the command line
- in the description file of the Maker
- in the ToolMaker common description file
- default setting

The above order is also the precedence of the occurrences. If an option is specified on the command line that specification overrides the values given in the description files. A value specified in the description file of the particular Maker overrides a specification in the common description file. And finally a value given in the common description file overrides built-in default values.

This scheme makes it easy to set values for a complete ToolMaker based subsystem (in the ToolMaker common description file), for a particular Maker (in the description file for that Maker) or for a single invocation of the Maker (on the command line).

7.4 Prefix Management

The prefix concept allows control over the naming of functions and type and enables adaption to various local customs and naming rules. All Makers have an option to set the prefix and it is also possible to set in the **.tmk** file.

The *system prefix* is used as a prefix for all system wide types and functions, i.e. functions and types that are defined by the ToolMaker subsystem as a whole and not particular to a specific Maker (e.g. the token type). A *Maker prefix* is the prefix used for functions and types defined by a particular Maker.

The relationships between these options and their values are the following:

- the default system prefix is '**tm**'
- if the prefix option is used in the ToolMaker Common Description file its value is used as the system prefix
- if no prefix is specified in the ToolMaker Common Description file, the default for the Makers are their local default ('**pm**', '**sm**' and '**lm**')
- if a system prefix is specified this is also the default for the Makers

This strategy makes it possible to use the prefix concept in a flexible and consistent manner.

7.5 Description File Sections

A section is a portion of a description file which may contain various kinds of information. There are two kinds of sections:

- named sections
- anonymous sections

Both kinds are opened by a pair of percentage signs ('%%'). Named sections also have their name attached to the percentage signs. An example of a named section is the Option section.

```
%%OPTIONS

    Target 'ansi-c';

%%END
```

Named sections are optionally closed explicitly by '%%END' or implicitly by the start of a new section. Text between sections are completely ignored.

Anonymous sections are for example the semantic sections in the grammar rules in a ParserMaker description file. There is no need to identify these sections explicitly.

7.6 Escape Character

As the percentage sign have special significance to the Makers a means to indicate that a percentage sign is not to be treated by the Makers is provided. The way to escape the special meaning of the % is to place a special character before the percent character. This special character is called the escape character. It is possible to set using an option in the description files and on the command line. See section *The Escape Directive* on page 32.

Note: The escape character is not used within strings, see below.

7.7 Strings

A string is a number of characters surrounded by single quotes ('), a quoted string. To place a single quote inside a string, use the backslash (\) to quote it. Strings are for example used as values for string valued options in the Option section and as message templates in the ListerMaker description file.

A THE PL/0 EXAMPLE

The following example is a ToolMaker subsystem, as described in the walk-through in *A TOOLMAKER BASED SYSTEM* on page 27, generated by ***toolmake*** for a SunOS (i.e. UNIX) environment and **'ansi-c'** as the target language. The subsystems name is **pl0** [Wirth] and contains ToolMaker-based scanner, parser and lister modules.

A.1 Directory Listing

The following listing shows the files that need to be maintained by the user from the PL/0 example.

```
-rw-r--r--  1 tools       1680 May 31 11:29 Makefile
-rw-r--r--  1 tools       3220 May 31 11:29 pl0.c
-rw-r--r--  1 tools        414 May 31 11:29 pl0.pmk
-rw-r--r--  1 tools        305 May 31 11:29 pl0.smk
-rw-r--r--  1 tools       1082 May 31 11:29 pl0.lmk
-rw-r--r--  1 tools       3474 May 31 11:29 pl0.tmk
```

This listing shows all the files for the complete PL/0 front end example, including objects and executable.

```
-rw-r--r--  1 tools       1680 May 31 11:29 Makefile
-rwxr-xr-x  1 tools      90112 May 31 11:30 pl0
-rw-r--r--  1 tools       3220 May 31 11:29 pl0.c
-rw-r--r--  1 tools        414 May 31 11:29 pl0.pmk
-rw-r--r--  1 tools        305 May 31 11:29 pl0.smk
-rw-r--r--  1 tools       1082 May 31 11:29 pl0.lmk
-rw-r--r--  1 tools       6294 May 31 11:30 pl0.o
-rw-r--r--  1 tools       3474 May 31 11:29 pl0.tmk
-rw-r--r--  1 tools         27 May 31 11:29 pl0.voc
-rw-r--r--  1 tools       6723 May 31 11:29 pl0Err.c
-rw-r--r--  1 tools       6751 May 31 11:30 pl0Err.o
-rw-r--r--  1 tools      38145 May 31 11:30 pl0List.c
-rw-r--r--  1 tools       2539 May 31 11:30 pl0List.h
-rw-r--r--  1 tools      36299 May 31 11:30 pl0List.o
-rw-r--r--  1 tools      45095 May 31 11:29 pl0Parse.c
-rw-r--r--  1 tools        647 May 31 11:29 pl0Parse.h
-rw-r--r--  1 tools      27681 May 31 11:30 pl0Parse.o
-rw-r--r--  1 tools       1653 May 31 11:29 pl0PaSema.c
-rw-r--r--  1 tools       1850 May 31 11:30 pl0PaSema.o
-rw-r--r--  1 tools      15131 May 31 11:30 pl0Scan.c
-rw-r--r--  1 tools        946 May 31 11:30 pl0Scan.h
-rw-r--r--  1 tools      17571 May 31 11:30 pl0Scan.o
-rw-r--r--  1 tools      17571 May 31 11:30 pl0ScSema.o
-rw-r--r--  1 tools      17571 May 31 11:30 pl0ScSema.o
```

A.2 pl0.tmk - the ToolMaker Description File

```
------------------------------------------------------------
-- pl0.tmk     Date: 1993-06-09/toolmake
--
-- Common ANSI-C definitions for all ToolMaker tools.
--
------------------------------------------------------------
```

```
-- Created:    1993-04-27/reibert@roo
-- Generated: 1993-06-09 14:48:30/toolmake v2,r0,c5
------------------------------------------------------------

%%OPTIONS
    Prefix 'pl0';

%%IMPORT
typedef int TmNatural;      /* A natural number. */
typedef int TmCode;        /* The type of a token code. */

%%END

%%SRCP Name TmSrcp;
    Row line %% TmNatural %1 %%;
    Column col %% TmNatural %1 %%;
    File file %% TmNatural %1 %%;
%%END

%%TOKEN Name TmToken;
    Code code %% TmCode %1 %%;
    Srcp srcp %% TmSrcp %1 %%;
    Attributes
        stringValue %% char %1[256] %%,
        integerValue %% int %1 %%;
%%END
```

A.3 pl0.c - the Main Program

```
/* pl0.c       Date: 1993-06-24/toolmake

   pl0 -- main program

   1.0 - 1993-06-24/
*/

#include <stdio.h>

#define PRIVATE static
#define PUBLIC

#ifndef FALSE
typedef int boolean;

#define FALSE 0
#define TRUE 1
#endif

#include "pl0Parse.h"
#include "pl0List.h"
#include "pl0Scan.h"


char *VERSION= "1.0";
char *NAME   = "PL/0 Analysis Tool";
char *USAGE= "Usage: pl0 [-h] [-l <name>] <in> [<out>]";

void summary();/* Imported from parser */


/* -- DATA -- */
```

```
PRIVATE char
    *listFileName= NULL,
    *inFileName= NULL,
    *outFileName= NULL;


/* -- SUBROUTINES -- */

/*-------------------------------------------------------
     perr()

     A simple writer of error messages (onto stderr),
     If 'addendum' is NULL it will not be written.
*/
PUBLIC void perr(
    char sev,
    char *desc,
    char *add
)
{
    if (add==NULL)
     fprintf(stderr, "%s: %c! %s\n", NAME, sev, desc);
    else
     fprintf(stderr, "%s: %c! %s: %s\n", NAME, sev, desc,
              add);

    if (sev=='F' || sev=='S') exit(1);
}


/* -- Argument processing -- */

PRIVATE void processArgs(
    int argc,
    char *argv[]
)
{
    int i;

    for (i=1; i<argc; ++i) {
        if (argv[i][0]=='-' && argv[i][1]) {
            switch (argv[i][1]) {
         case 'l':
          listFileName= argv[++i];
          break;
         case 'h':
          printf("%s\n\n", USAGE);
          printf("Options:\n");
          printf("  -l <name>  Set list file name\n");
          exit(0);
         default:
          perr('W', "unknown switch", argv[i]);
          break;
         }
     }
     else if (inFileName==NULL) inFileName= argv[i];
     else if (outFileName==NULL) outFileName= argv[i];
     else perr('W', "unknown argument", argv[i]);
    }
}


/* -- MAIN -- */
```

```
PUBLIC main(
    int argc,
    char *argv[]
){
    boolean stdIn;

    processArgs(argc, argv);
    printf("%s - %s\n\n", NAME, VERSION);

    if (inFileName == NULL)
        perr('F', "No input file", NULL);

    stdIn= strcmp(inFileName,"-")==0;
    if (stdIn)
        pl0LiInit(VERSION, "standard input",
                  pl0_ENGLISH_Messages);
    else
        pl0LiInit(VERSION, inFileName,
                  pl0_ENGLISH_Messages);
    if (!pl0ScanEnter(inFileName)) {
        pl0Log(NULL, 199, sevFAT, inFileName);
        pl0List("", 0, 78, liTINY, sevALL);
    } else {
        pl0Parse();
        pl0ScanTerminate();
        if (stdIn) {/* This gives an idea of processing of
                       error messages */
         int i;
         char err[1024];
         TmSrcp srcp;

         /* Print a list on the terminal if any errors */
         for (i=1; pl0Msg(i, &srcp, err); i++)
          printf("\"-\", line %d: %s (column %d)\n",
                 srcp.line, err, srcp.col);
     } else {
         /* Print a TINY list on the terminal */
         pl0List("", 0, 78, liTINY, sevALL);
            summary();
         /* And a list file if asked to. */
         if (!stdIn && listFileName != NULL) {
          pl0List(listFileName, 60, 132, liFULL, sevALL);
                summary();
            }
     }
    }
    pl0LiTerminate();
}
```

B ERROR MESSAGES

B.1 Message Format

Error messages generated from the Makers has the form:

```
['*' <sequence number> '*'] <error number> <severity>
                            ':' <error text>

<sequence number> ::= <number>

<error number> ::= <number>

<severity> ::= 'I' | 'W' | 'E' | 'F' | 'S'

<error text> ::= {<character>}
```

If an error appears during the analysis of the input, the message is preceded by the numbered source line containing the error. The erroneous symbol is clearly marked by the `sequence number` in the same position as the offending symbol on the line below the source line.

The `error number` is used to uniquely identify the error message.

The error `severity` indicates the seriousness of the message. The severity codes have the following meaning:

`I` = *Informational message*. Used to inform the user that ParserMaker has taken some action. The only reason to correct the "error" is to get rid of the message since the Maker will always produce a correct result. These messages can also be inhibited by excluding the option `Info` from the `List` directive.

`W` = *Warning*. A warning is not an error as such, but merely an information to the user that something may have gone wrong. The Maker always creates a valid result, but it may not always be what the user intended.

`E` = *Error*. Normally a syntactic error. The Maker tries to recover from the error and continue the current phase of the processing. The process will be terminated after the current phase. No valid result is constructed.

`F` = *Fatal error*. An error that it impossible to recover from. The processing is immediately terminated, and no valid result is produced.

`S` = *System error*. An internal Maker error. The processing is immediately terminated, and no valid result is produced. Please save the input files that caused the error and contact your ToolMaker contact person.

The `error text` is a verbal description of the error that has occurred. In the description below, the text may contain the character sequence `%n` where

n is a number. This is a placeholder for a variable string that will be inserted when the message is actually output.

B.2 Messages Explanations

A number of error messages are common for all Makers. These are numbered between 50 and 100 and concerns analysis of the option, token and srcp sections. These are described below.

50 `Incorrect option.`

The option specified is not defined in the Maker.

51 `Incorrect section.`

The section is not defined to be used in the Maker.

52 `%1 section not allowed in this file when .tmk file exists`

This section may only be used in **.tmk** when there exist such a file.

60 `Section defined twice, new section skipped.`

This section can only be specified once.

61 `Directive defined twice, new directive skipped.`

A directive can only be specified once.

62 `Token section must have a %1 member.`

The token must have the specified structure member field.

63 `Srcp section must have a %1 member.`

The source position must have the specified structure member field.

64 `%1 section not found.`

The required section is missing.

65 `If declaration is given in one %1 member it must be given in all.`

When declaring a token or source position all structure member fields must have a declaration or else none must have a declaration. A structure member field may not have a declaration if there are others that do not.

66 `Srcp section must have at least one member.`

Source position must at least have one of its structure member fields specified. ScannerMaker and ListerMaker also requires that if a column field is specified, a row field must also be specified.

`70 Could not open input file "%1".`

The input file could not be found or was read protected.

`71 Could not open output file "%1".`

The output file could not be opened, the directory or a previous file with the same name was probably write protected.

B.3 License Errors

The following errors are due to license limitations or problems and are only output to the terminal. The processing is terminated immediately.

`500 F License server: date expired`

Your ToolMaker license has expired. Contact your ToolMaker contact person.

`501 F License server: no license available`

You were not able to get a ToolMaker license as there were too many other simultaneous users. Try again later.

`502 F License server: format error`

The license file did not have the correct format. Contact your ToolMaker contact person.

`503 F License server: no contact`

ToolMaker was not properly installed or your license server process has died. Contact your system administrator or ToolMaker contact person.

`504 F License server: license file missing`

No license file was found in the **TMHOME** directory. Contact your ToolMaker contact person.

`505 F License server: illegal license key`

The password in the license file was illegal. Contact your ToolMaker contact person.

`506 F License server: unknown error`

Contact your local ToolMaker contact person.

C SYNTAX NOTATION

The syntactic structure of the description files throughout this documentation are described using a modified EBNF. This appendix briefly describes that notation. This syntax is used to describe the structure of the description files in a ToolMaker-based system, it is *not* the syntax you should use for your context free grammar productions in the ParserMaker description file (see *The Rules Section* in the *ParserMaker Reference Manual*, page 99 for a description of that syntax).

Words enclosed by angle brackets denote syntactic categories (nonterminals). Example:

```
<terminal definition>
```

Upper case words denote pseudo terminals. Example:

```
IDENTIFIER
```

Quoted strings denote keywords and other terminal symbols. Example:

```
';' 'RECOVERY' '%%OPTIONS'
```

The symbol '::=' should be read as "is defined as". Example:

```
<right hand side> ::= <components> <opt modify>
```

Curly braces denotes repetition. The items within braces are repeated zero or more times. Example:

```
<number> ::= <digit> {<digit>}

<terminal list> ::= TERMINAL {',' TERMINAL}
```

Square brackets enclose optional items. Example:

```
<verbose directive> ::= ['NO'] 'VERBOSE' ';'

<terminal definition> ::=
     TERMINAL '=' <token code> [<error recovery data>]
```

A vertical bar separates alternative items. Example:

```
<option> ::= <recovery option>
           | <optimize option>
```

Parentheses are used to hierarchically group concepts together. Example:

```
<opt modify> ::=
     {('%+' | '%-') '(' TERMINAL {',' TERMINAL} ')'}
```

D REFERENCES

[Aho] Aho A.V., Sethi R., Ullman J.D. - "Compilers Principles, Techniques and Tools", Addison-Wesley 1986.

[Backus] Backus et al - "Revised Report on the Algorithmic Language ALGOL 60", International Federation for Information Processing 1962.

[Dastranj] Dastranj-Sedghi A. - "A Transformational Approach for Handling Extended BNF Grammars in PWS", Linköping University, LiTH-IDA-Ex-8932, 1989.

[Gries] Gries D. - "Error Recovery and Correction - An introduction to the literature", In Bauer F.L. (ed) "Compiler Construction", Lecture Notes in Computer Science 21, Springer Verlag, 1976, pp 627-638.

[KR] Kernighan B.W, Ritchie D.M. - "The C programming language", Prentice-Hall Inc. 1983.

[Sencker] Sencker P., Durre K., Heuft J. - "Optimization of Parser Tables for Portable Compilers", acm Transactions on Programming Languages and Systems, Oct. 1984, Vol.6 No. 4, pp. 546-572.

[Sippu] Sippu S. - "Syntax error handling in compilers", PhD Thesis, Department of Computer Science, University of Helsinki, 1981.

[Wirth] Wirth, Niclaus - "Algorithms + Data Structures = Programs", Prentice-Hall, 1976.

Part II

ParserMaker
Reference Manual

1 INTRODUCTION 59

2 CONCEPTS AND ASSUMPTIONS 60

2.1 Backus-Naur Form (BNF) 60
2.2 Extended Backus-Naur Form (EBNF) 60
2.3 Productions 62
2.4 Semantic Actions 63
2.5 Grammar Attributes 64
2.6 Grammar Ambiguity and LALR-conflicts 68
2.6.1 Default Disambiguating Rules 71
2.6.2 Modification Rules 71
2.7 Error Recovery Principles 72
2.7.1 Level 1: Single Symbol Correction 73
2.7.2 Level 2: String Synthesizing Technique 74
2.7.3 Level 3: Panic Mode 75
2.7.4 Improvements of the Error Recovery System 75

3 PARSER PRODUCTION 76

3.1 Creating a Running Parser 76
3.1.1 File Descriptions 77
3.2 The ParserMaker Description File 79
3.2.1 Lexical Definitions 79
3.2.2 Overall Structure of the Description File 83
3.2.3 An Example 84
3.2.4 The Options Section 86
3.2.5 The Import, Export and Declarations Sections 93
3.2.6 The Terminals Section 94
3.2.7 The Attributes Section 95
3.2.8 The Scanner Section 96
3.2.9 The Insertsymbol Section 97
3.2.10 The Deletesymbol Section 97
3.2.11 The Recovery Section 97
3.2.12 The Rules Section 99
3.3 The ToolMaker Common Description File 100
3.4 The List File 100
3.4.1 Format of the Pretty Printed Grammar 100
3.4.2 Format of the Generated Item Set 100
3.4.3 Format of the Parser Tables 102
3.5 The Vocabulary File 102

4 THE PARSERMAKER COMMAND 104

4.1 Parameters 104
4.2 Options 104

4.3 Example 105

5 PARSER RUN-TIME USAGE 106

5.1 Principles of Operation 106
5.1.1 Parsing 106
5.1.2 Scanning 106
5.1.3 Error Recovery 106

5.2 Run-Time Interface 107
5.2.1 Function: *pm*Parse() 107
5.2.2 Function: *pm*ISym() 108
5.2.3 Function: *pm*DSym() 108
5.2.4 Function: *pm*RPoi() 109
5.2.5 Function: *pm*Mess() 109

A Appendix: THE PL/0 EXAMPLE 111

A.1 pl0.pmk - The ParserMaker Description File 111

B Appendix: ERROR MESSAGES 117

B.1 Message Explanations 117

C Appendix: DESCRIPTION LANGUAGE 123

C.1 Language Syntax 123
C.2 Lexical Items 126

D Appendix: TARGET LANGUAGE DETAILS .. 128

D.1 'c' 128
D.2 'ansi-c' and 'c++' 128

1 INTRODUCTION

ParserMaker is a general and advanced tool for the construction and maintenance of language processors - parsers. ParserMaker system contains many features which makes it easy to use for various applications, facilities that are not common in similar systems.

General features of ParserMaker are:

- It generates highly optimized parsers for different target languages.
- It automatically generates a three level error recovery system from the grammar specification.
- It supports two different techniques for handling ambiguous grammars.
- It supports the creation of a system by a special description file and the possibility of inserting target dependent code in the input (semantic actions).
- It contains a mechanism for the support of synthesized attributes in the semantic actions.
- It is highly integrated with the other tools in SoftLab's ToolMaker.
- It supports multiple target languages, currently 'c', 'ansi-c' and 'c++'.

The input description consists of a set of context free grammar productions, extended with target dependent code segments. Output from ParserMaker consists of one or more complete programs recognizing the language specified as input.

This part of the ToolMaker documentation contains no description of the theory behind ParserMaker. Knowledge about parsing theory, especially concerning LR-parsers, is required in order to fully understand the document. Familiarity with the other tools in the ToolMaker family is also beneficial but not necessary.

2 CONCEPTS AND ASSUMPTIONS

2.1 Backus-Naur Form (BNF)

A commonly used technique for describing the grammar of a language is the *Backus-Naur form* (BNF for short), so called after its inventors (see reference [Backus]). A grammar described in BNF consists of a number of "metalinguistic variables" represented by characters enclosed in angle brackets '<>', the symbol '::=' which is interpreted as "is defined as", the symbol 'I' which is interpreted as "or" and all other symbols which stand for themselves. For example, the IF-statement in Pascal may be defined as:

```
<if statement> ::=
      IF <expression> THEN <statement>
    | IF <expression> THEN <statement> ELSE <statement>
```

The ParserMaker input description language is closely related to BNF, but differs in some details. The IF-statement in the ParserMaker description language could look like:

```
if_statement =
      'IF' expression 'THEN' statement
    ! 'IF' expression 'THEN' statement 'ELSE' statement
    ;
```

There are a number of variations possible. The differences can be summarized as follows:

- '=' is used instead of '::='.
- '!' is used instead of 'I'.
- ';' is used to indicate the end of a rule.
- Metalinguistic variables (non-terminals) may be written as identifiers or as characters enclosed in angle brackets.
- All other symbols (terminals) may be written as identifiers, as characters enclosed in angle brackets or as single-quoted strings.

2.2 Extended Backus-Naur Form (EBNF)

Although the BNF form is fairly expressive, it suffers from a number of shortcomings, for example it generates a lot of rules. To remedy these shortcomings a number of extensions to the BNF notation has been proposed which has resulted in the *Extended Backus-Naur Form* (EBNF for short). These extensions are:

1) *Repetition* of symbols within a rule.

2) *Nested alternation* within a rule.

3) *Optional symbols*.

Repetition of symbols is indicated by the symbols being enclosed in *curly braces*, "{}", which indicates that the enclosed symbols may be repeated zero or more times. For example, in BNF a non-empty statement list would be described by the two rules:

```
<statement list> ::= <statement list> <statement>
                   | <statement>
```

In EBNF a single rule is sufficient:

```
<statement list> ::= <statement> {<statement>}
```

Nested alternation is expressed by enclosing related symbols within *parentheses*, and separating the alternatives with a *vertical bar,* '|'. For example, an expression grammar in BNF could be described by the rules:

```
<expression> ::= <expression> <addop> <term>
               | <term>

<term> ::= <term> <mulop> <factor>
         | <factor>

<addop> ::= + | -

<mulop> ::= * | /
```

In EBNF we would write:

```
<expression> ::= <expression> (+ | -) <term>
               | <term>

<term> ::= <term> (* | /) <factor>
         | <factor>
```

Or even simpler:

```
<expression> ::= {<term> (+ | -)} <term>

<term> ::= {<factor> (* | /)} <factor>
```

Optional symbols are enclosed in *square brackets*, '[ ]', which indicates that the enclosed symbols may occur once or not at all. For example, the Pascal IF-statement may be described by:

```
<if statement> ::= IF <expression> THEN <statement>
                   [ELSE <statement>]
```

In the ParserMaker input description language, repetition and nested alternation are supported. The EBNF symbols used in the ParserMaker input description language are:

1) *Curly braces* '{}' are used to denote *repetition.*

2) *Parentheses* '()' and the *vertical bar* 'I' are used to express *nested alternation.*

3) *Square brackets* '[ ]' are used to delimit *optional symbols.*

2.3 Productions

A pure ParserMaker grammar production has the form:

```
<left> = <rhs>;
```

where `<left>` is a nonterminal and `<rhs>` is the right hand side. The right hand side consists of a sequence of zero or more elements where each element is a terminal or nonterminal symbol. If a symbol has more than one right hand side the symbol "!" can be used to separate the rules.

Example: Writing:

```
list = list ',' element
     ! element
     ;
```

is equivalent to writing:

```
list = list ',' element;

list = element;
```

A production or rule with no right hand side is called an *empty* or *null production.* Thus, one might write:

```
left = ;
```

It is also possible to include the EBNF constructs in the productions in which case an element can be, apart from a terminal and a nonterminal, an alternation construct or a repetition construct. An alternation construct consists of one or more alternatives enclosed in parentheses, where each alternative is a sequence of elements, and the alternatives are separated by vertical bars. A repetition construct consists of a sequence of elements enclosed in curly braces. For example, a list may be written as:

```
list = element {',' element};
```

using the repetition construct. If we allow a semicolon, as well as a comma, to act as a separator in our example list, this could be written as:

```
list = element {(',' | ';') element};
```

simply and elegantly in a single rule.

One of the nonterminal symbols is especially important, namely the *goal symbol*. The entire language is derived from that particular symbol. The ParserMaker system determines the goal symbol using the following rules:

- If there is one nonterminal that never appears in any right hand side of the grammar that particular symbol is chosen. If more than one symbol fulfils this rule the grammar is erroneous.
- If no symbol can be found satisfying the above rule, the left hand side of the first production is chosen as the goal symbol.

ParserMaker requires a grammar with a unique goal symbol, and creates an augmented grammar by adding the production:

```
ParserMaker = goal_symbol $;
```

where `ParserMaker` is the unique system generated symbol, `goal_symbol` is the found goal symbol and `'$'` is the end marker.

2.4 Semantic Actions

Semantic actions or *semantic rules* are code written in the target language which is evaluated when a particular production is reduced by the parser. The semantic action mechanism is the easiest way to attach executable code (written by the parser implementor) to the parser. The semantic actions are written in the target language. In ParserMaker no checks of the target dependent code are made, this is deferred until the generated parser source files are compiled using the appropriate target language compiler.

A semantic action is normally inserted after the grammar rule. This is due to the fact that the parser can only invoke the semantic action when an entire production has been recognized. An action consists of arbitrary code between a pair of '%%'-symbols.

Example:

```
left = rhs
          %%
               printf("rhs is identified");
          %%
     ;
```

When `rhs` has been identified, the output action is executed and the desired text is written. Variables and functions referenced within a semantic action must be defined in the *declaration section* or imported using target language code in the *import section* (see *The Import, Export and Declarations Sections* on page 93) in order for the generated parser to compile correctly.

ParserMaker also supports actions within the right hand side and not necessarily at the very end. In order to get hold of the action during parsing, a new empty production is created to control the execution.

Example: The production with attached actions:

```
list = element
          %% nr_elements = nr_elements + 1; %%
     ',' list
     ;
```

is converted to:

```
list = element <genSym> ',' list;

<genSym> = %% nr_elements = nr_elements + 1; %%;
```

When `element` has been recognized, the dummy production `<genSym>`, which is a system generated name, is immediately reduced (on ','), and the appropriate action is called.

Warning! This way of inserting a semantic action within the right hand side may violate the LALR(1) conditions. However, a semantic action at the end will never interfere with the generation process.

2.5 Grammar Attributes

ParserMaker supports a technique often denoted by the term *attributed translation grammar* or *attribute grammar*. This means that a semantic action may propagate values which may be retrieved by semantic actions in other productions.

Within a semantic action one can write:

```
%symbol.attribute
```

which means that the value denoted by `attribute` belonging to the current instance of `symbol` is referenced. `Symbol` is a grammar symbol, either a terminal or a nonterminal, appearing in the production to which the semantic action is attached.

Example: The rule:

```
left = rhs
          %%
               %left.VAL = 10;
          %%
     ;
```

means that the result from the action is 10, and this value is connected to the symbol `left` and denoted by the attribute value `VAL`.

Example: To retrieve values from descendants:

```
left = x y z
          %%
               %left.RES = %x.VAL + %y.VAL + %z.VAL;
          %%
     ;
```

The `VAL` values of `x`, `y`, and `z` are added, and the result is attached to `left` and bound to the attribute `RES`. The values of `VAL` have presumably been defined in the definition rules of `x`, `y` and `z` respectively.

If a production contains more than one symbol with the same name, the symbol must be qualified. *Qualification* is performed by numbering the symbols.

Writing:

```
%<n>symbol
```

means the `nth` occurrence of the `symbol`. `%1symbol` is equivalent to `%symbol`.

Example:

```
expr = expr + term
          %%
               %expr.VAL = %2expr.VAL + %1term.VAL
          %%
     ;
```

The resulting value which is assigned to the left hand side `expression` is achieved by adding the second `expression` to the `term`.

When using semantic actions in EBNF rules, there exist two meta-symbols which allow groups of symbols (corresponding to an EBNF construct) to have attributes. These symbols are `EBNF` which refers to an EBNF construct preceding the attribute reference, and `OEBNF` (short for Outer EBNF) which refers to the closest enclosing EBNF construct. If there are several EBNF constructs preceding a semantic action containing attribute references using the `EBNF` symbol, it is possible to access all of them using *qualification* as usual. If the closest enclosing EBNF construct is a repetition, it is possible to use the symbol `1OEBNF` to refer to the attributes of the current instance of the repe-

tition, and the symbol 2OEBNF to refer to previous instances of the repetition. When using repetition, a semantic action immediately following the left curly brace is used as an *initializing action*, whereas a semantic action immediately preceding the right curly brace is used as a *repetitive action*. Consider the example:

```
a = b {         %% /* Initializing action */
                     %OEBNF.val = 0;
                %%
       c
        (d      %% %OEBNF.val = %d.val; %% --OEBNF='(d|e)'
          | e   %% %OEBNF.val = %e.val; %%
        )       %% /* Repetitive action */
                     %1OEBNF.val = %2OEBNF.val +
                          %c.val + %EBNF.val;
                %%   -- 1OEBNF=this instance of '{..}'
                     -- 2OEBNF=previous instance of '{..}'
                     -- EBNF='(d|e)'
     } f        %% %a.val =
                     %b.val + %EBNF.val + %f.val;
                %%   -- EBNF='{..}'
    ;
```

This complex rule is transformed to the following sequence of productions:

```
a = b <genSym1> f
          %% %a.val =
               %b.val + %<genSym1>.val + %f.val;
          %%
    ;

<genSym1> = -- Empty
               %% /* Initializing action */
                    %<genSym1>.val = 0;
               %%
          ! <genSym1> c <genSym2>
               %% /* Repetitive action */
                    %1<genSym1>.val = %2<genSym1>.val +
                         %c.val + %<genSym2>.val;
               %%
          ;

<genSym2> = d
               %% %<genSym2>.val = %d.val; %%
          ! e
               %% %<genSym2>.val = %e.val; %%
          ;
```

In the derived rules, <genSym1> corresponds to the repetition construct, and <genSym2> corresponds to the alternation construct. Note the correspondence of the EBNF and OEBNF symbols to the generated symbols. Note also where the initializing action and the repetitive action are placed.

ParserMaker supports only *synthesized attributes*. This means that computations within an action are only dependent of the right hand side symbols and not on the left hand side symbol. Attributes are propagated from the leaves (terminals) towards the goal symbol. However using an explicit stack and em-

bedding push and pop semantic actions within the rules a limited form of inherited 'attributes' may be implemented.

Semantic actions embedded in a production (i.e. not located at the end) are only permitted if the actions are not referencing any attributes. Exceptions are EBNF rules where embedded semantic actions may contain attribute references to the symbols in the current EBNF context.

Lexical attributes, attributes connected to terminal symbols, must not necessarily be the same as the nonterminal attributes. The two types are declared separately in the *attributes section*, see *The Attributes Section* on page 95.

> *Note:* To simplify the propagation of attributes, the left hand side symbol shares its attributes with the first symbol on the right hand side if this is a nonterminal. This means that by default, all the attributes of the first symbol on the right hand side will be propagated automatically to the left hand side symbol. Specifically it means that no explicit propagation has to be done for unit productions (e.g. `expression = term`). However, with this mechanism care must be taken not to modify an attribute value of the left hand side symbol before the corresponding attribute value of the first symbol on the right hand side symbol is used. Again, this mechanism applies only if the first symbol on the right hand side is a nonterminal.

Example: The complete specification below defines a small grammar for additive expressions. The lexical symbol NUMBER defines an attribute named SCAN_VALUE. The grammar defines the `expression`, computes its value in the 'c' language, and outputs it.

```
%%TOKEN
     NAME TokenType;
     CODE code %% int code %%;
     SRCP srcp %% int srcp %%;
     ATTRIBUTES
          SCAN_VALUE %% int SCAN_VALUE %%;

%%ATTRIBUTES
     VAL %% int VAL %%;

%%RULES

expression
     = expr
          %% printf("VALUE: &%i\n", %expr.VAL); %%
     ;

expr
     = expr '+' term
          %% %1expr.VAL = %2expr.VAL + %term.VAL; %%
     ! term
          %% %expr.VAL = %term.VAL; %%
     ;
```

```
term
     = '(' expr ')'
          %% %term.VAL = %expr.VAL; %%
     ! NUMBER
          %% %term.VAL = %NUMBER.SCAN_VALUE; %%
     ;
```

The target language type of an attribute is application dependent, and it is up to the parser implementor to define the attribute type (see also *The Attributes Section* on page 95).

2.6 Grammar Ambiguity and LALR-conflicts

For some input grammars ParserMaker is not able to create a consistent parser. This is dependent on the fact that the input grammar is either ambiguous or is a non-LALR(1) grammar. An *ambiguous grammar* can derive an input string in at least two different ways. A grammar can be unambiguous but still not LALR(1) because a lookahead longer than one (1) symbol is required.

An LALR(1) grammar can also be characterised as a grammar for which an LALR(1) parser can be constructed with a unique action for each terminal symbol and state. If this is not possible the parser contains at least one *LALR(1) conflict*. The conflicts can be either of two kinds - shift-reduce or reduce-reduce conflicts.

In a *shift-reduce conflict* the parser can not determine whether to shift a symbol on the parse stack or replacing some portion of the stack with a nonterminal (reduce).

In a *reduce-reduce conflict*, at least two equivalent right hand sides can be applied in a reduce action. The parser can not determine which one to apply.

A conflict is an indication of that something is wrong in the input grammar. In most cases the conflicts are resolved by rewriting the grammar slightly. However, there are situations when this approach is very troublesome. For such occasions ParserMaker contains mechanisms called *disambiguating rules* which makes it possible to create a parser even for certain ambiguous grammars.

ParserMaker contains two such mechanisms:

- A *default technique* which requires no assistance from the parser implementor.
- A technique called *modification* which directly modifies the parse tables.

Before describing the disambiguating rules used in ParserMaker, two well-known problems will be discussed - the dangling else and the ambiguous expression.

Ambiguous grammar 1: The dangling else

Consider the productions for the well-known IF-statement:

```
<if statement>
     = 'IF' <condition> 'THEN' <statement>
     ! 'IF' <condition> 'THEN' <statement>
       'ELSE' <statement>
     ;

<statement>
     = <if statement>
     ! ...
     ! ...
     ;
```

It can easily be seen that the above grammar is ambiguous by the statement:

```
IF c1 THEN IF c2 THEN s1 ELSE s2
```

where `c` and `s` stands for condition and statement respectively. The ambiguity occurs because there is no way of determining to which IF the ELSE-clause belongs. It can be bound to the innermost THEN representing the structure:

```
IF c1 THEN
     IF c2 THEN s1
     ELSE s2
```

or to the outermost THEN giving:

```
IF c1 THEN
     IF c2 THEN s1
ELSE s2
```

The ParserMaker system will recognize the conflict in a state having the LALR(1) items:

```
<if statement> -->
     IF <condition> THEN <statement> .
          { ... 'ELSE' ... }

<if statement> -->
     IF <condition> THEN <statement> . ELSE <statement>
```

The grammar contains one shift-reduce conflict. ParserMaker does not know whether to shift the symbol ELSE, binding the ELSE-clause to the innermost THEN, or to reduce and bind the ELSE to an outer THEN-clause.

The normal interpretation of an IF statement is to bind the ELSE to the innermost THEN-clause. Thus, shifting the symbol ELSE will do the job.

Ambiguous grammar 2: The ambiguous expression

```
<expr> = <expr> '+' <expr>
       ! <expr> '*' <expr>
       ! I
       ;
```

The above grammar contains a number of LALR(1) conflicts. The conflicts in this grammar are associated with operator associativity and precedence.

The expression:

```
I + I + I
```

can be interpreted either being left or right associative, yielding the structures:

```
(I + I) + I
```

or

```
I + (I + I)
```

respectively.

The system gets a shift-reduce conflict when recognizing the input `I + I`, representing `<expr> + <expr>`, and the next token is '+'. ParserMaker can then not determine whether to shift, interpreting '+' to have right associativity, or reduce, getting left associativity.

The most common interpretation of additive expressions is to let '+' and '*' be left associative. The conflict is solved by reducing `<expr> + <expr>` when '+' is the look-ahead symbol.

The second type of conflict arises when:

```
I + I * I
```

or

```
I * I + I
```

appears as input. If we consider the first one it could be structured into:

```
I + (I * I)
```

or

```
(I + I) * I
```

The first interpretation defines '*' to have higher precedence than '+' and vice versa for the second one. This situation is quite analogous to the associativity problem above. If the input is `I * I` and the next token is '+', the action must be to reduce `<expr> * <expr>`. But, if the input is `I + I` and the next token is '*', the action must be shift.

As already mentioned, ParserMaker contains two different mechanisms to help producing a consistent parser. Again, the safest way to avoid problems is to rewrite the grammar, or, if possible, to modify the language.

2.6.1 Default Disambiguating Rules

If a grammar is not LALR(1), and no modifications resolve the conflict, a default disambiguating mechanism will be used. A conflict is by default resolved by the rules:

- In a *shift-reduce* situation, use shift in favour of reduce.
- A *reduce-reduce* conflict is resolved by reducing the production that comes first in the input grammar.

The dangling else problem is solved adequately by applying the first rule. That is, the ELSE will always be shifted and bound to the innermost THEN.

The ambiguous expression however, will not be solved correctly since different look-ahead symbols are treated differently, and it is impossible to predict the logic of every such situation.

2.6.2 Modification Rules

The modification rules directly affects the parse tables. The mechanism requires great attention and should be used with care.

A conflict is resolved by preventing or forcing a reduce action on specified look-ahead symbols. The modification rules are written:

```
%+ (<terminals>)
```

or

```
%- (<terminals>)
```

`%+` means *reduce-for* the indicated symbols, while `%-` means *reduce-not-for* the indicated symbols.

The modification rules can appear at two different places in a right hand side:

- At the end of a grammar rule.
- Immediately before and/or after an embedded semantic action. The modifications are then bound to the generated dummy production.

When an input grammar contains LALR(1) conflicts, the affected states are written to a list file. The conflicting look-ahead symbols can then be inspected and appropriate actions can be decided.

As examples of the modification mechanism, consider the solutions to the dangling else and the expression problems. The dangling ELSE problem is solved by avoiding a reduce for the look-ahead symbol ELSE. This is written:

```
<if statement>
     = 'IF' <condition> 'THEN' <statement> %-('ELSE')
     ! 'IF' <condition> 'THEN' <statement>
       'ELSE' <statement>
     ;
```

An unambiguous grammar for the expression is specified below. The interpretation is that '*' has higher precedence than '+', and both operators are left associative.

```
<expr> = <expr> '+' <expr> %+('+') %-('*')
       ! <expr> '*' <expr> %+('+', '*')
       ! I
       ;
```

A modification is considered by ParserMaker only when a conflict is resolved.

When a conflict occurs, the following actions should be taken:

- In a *shift-reduce* situation, the conflict is resolved by specifying an action for the conflicting look-ahead symbol of the reduced production.
- In a *reduce-reduce* conflict, modifications must be specified for the conflicting symbol in both productions.

2.7 Error Recovery Principles

A *syntax error* is detected when the parser does not accept the next token. The task of the generated error recovery system is to recover from the error. *Error recovery* means that the parser configuration, i.e. state and input, is changed in such a way that parsing can be resumed. Whenever possible ParserMaker will use the approach *recovery by repair*. It tries not only to restart the parser, but also to correct the error. The term *error correction* is adopted for this proc-

ess. The general approach is to find as many syntax errors as possible in a single run.

The generated error recovery system in ParserMaker is advanced and efficient. It requires very little extra information besides the grammar itself. The extra information needed is grammar dependent, and is aimed for tuning purposes only. The error recovery system uses a three level approach. These levels are:

1) *Single* symbol correction.

2) *Multiple* symbol correction by a string synthesizing technique.

3) *Panic* mode.

In the normal case these levels or phases are applied in this particular order. Starting with the single symbol corrector and continuing with the string synthesizing corrector if the first level fails. Finally, if the second level fails the last one, panic mode, is applied. However, by options (see *The Options Section* on page 86) any level can be turned on or off. Each one of the error recovery phases is described below.

2.7.1 Level 1: Single Symbol Correction

The first level is a *restricted local corrector*. Restricted because it is not able to recover from all types of syntax errors. Local correction means that it will change only the remaining input and not the already parsed text. The first level tries to find a least cost correction of single symbol errors. One symbol of insertion, deletion or replacement will be considered.

In the error configuration of the parse stack and input, each possible single symbol repair is considered, and the correction with the least cost is applied. The cost is based on the costs assigned to each terminal in the *terminals section*, and how well the correction fits in the context in which the error occurred.

The cost of the best repair is compared with a predefined threshold value. If the computed cost is below this threshold value, the correction is applied. If not, the first level fails. By default the threshold value is equal to the cost of accepting two symbols after the error symbol.

The acceptance cost and the amount of look-ahead are defined using options. (see *The Options Section* on page 86). Together with the facility of defining costs for each terminal symbol (see *The Terminals Section* on page 94) a parser implementor has an opportunity to tune the first level of recovery to a specific environment.

The *acceptance (shift) cost* has a default value of 5. This cost should be regarded as an upper bound for the insert and delete costs for the terminal sym-

bols. It is only necessary to change the acceptance cost if you want the terminal costs to have a greater range.

The *amount of look-ahead* has a default value of 5. This value is a reasonable trade-off between functionality and performance. Since the threshold value is defined as *(look-ahead-amount - 3) * shift-cost*, you are not allowed to reduce the amount of look-ahead below 4. The threshold value would in that case get a value of less than or equal to zero which means that the single symbol correction will always fail. Due to performance reasons, it is not recommended to increase the look-ahead above 6.

Defining appropriate cost functions is not easy and should be based on some statistical observations on how the language is used. The general idea behind the cost functions is that they should be selected such that the lower the probability of a specific error occurring, the higher the cost should be for the symbol correcting the error. For instance, omission of a semicolon is a very frequent error in Pascal, thus the insertion cost for ';' should be low. Similarly long reserved words should be assigned high deletion costs.

It can sometimes be useful to deliberately violate the rule that the acceptance cost should be regarded as an upper limit of the terminal costs. If you want to make sure that a specific terminal symbol is never inserted or deleted during error correction, you should give it an insertion or deletion cost which is larger than the threshold value.

2.7.2 Level 2: String Synthesizing Technique

If the first level of recovery fails, the parser will enter the second phase. The string synthesizer will, from the error point, generate a least-cost string which yields a legal continuation of the already parsed text. The least cost string is generated from the grammar and based on the insertion costs (defined in *The Terminals Section* on page 94).

The generated string is then matched against the remaining input. In this process, a prefix of the generated string will be inserted, and some symbols from the remaining input might be deleted. The idea is to preserve as much as possible of the remaining input and base the recovery on insertion rather than deletion.

The string synthesising technique is a *true local corrector* as it never gives up. It can return a legal configuration for each possible input string. However, the parser will terminate the second level when it tries to delete a *fiducial symbol* (however see the modified panic mode algorithm below). A fiducial symbol is a syntactically important symbol, and it is defined as fiducial by the parser implementor (see *The Fiducial Part* on page 98). Symbols that start main phrases of the input language, e.g. some of the reserved words like `VAR`, `BEGIN` and `PROCEDURE` in Pascal, should be defined as fiducial symbols.

2.7.3 Level 3: Panic Mode

Panic mode is a very crude way of recovering from syntax errors. The general idea behind panic mode is to scan ahead into the input string until something solid, like a reserved word, is found. The parse stack is then reconfigured, popped, such that the new token is accepted. The term *fiducial* will be used for such important symbols. See *The Fiducial Part* on page 98 for how to specify these symbols.

In the original panic mode large portions of the remaining input might be deleted, and ParserMaker therefore uses a modified variant.

The technique used by ParserMaker can be summarized by:

1) Scan input until a fiducial symbol is found. If panic mode is entered from level 2, a fiducial symbol is already the current token.

2) Pop the parse stack until a state is found that will accept the fiducial symbol.

3) If in step 2 no such state is found then
restore stack
if entered from level 2 then
resume level 2 where it was terminated
else continue with step 1 above

Even if panic mode is a crude way of recovery, it might be the only reasonable choice when dealing with gross structural syntax errors.

2.7.4 Improvements of the Error Recovery System

The error recovery system is generated almost entirely from the grammar, and very little extra information must be supplied by the parser implementor. There are however, several ways of enhancing the generated error recovery system. Parser implementor supplied information can be added in the *terminals section* (see *The Terminals Section* on page 94), by the cost functions and print symbol, and in the *recovery section* (see *The Recovery Section* on page 97). The *recovery section* contains facilities for defining the set of fiducial symbols and some other tuning information. Also, the shifting cost and the amount of look-ahead applied can be defined using options to ParserMaker (see *The Options Section* on page 86)

3 PARSER PRODUCTION

3.1 Creating a Running Parser

The figure below gives an overview over the relation between the files required and produced by ParserMaker. The prefix ***pm*** is chosen by the parser implementor and is dependent on the system prefix and the parser prefix (see *The Options Section* on page 86), and the file names for any produced target source code module is target language dependent.

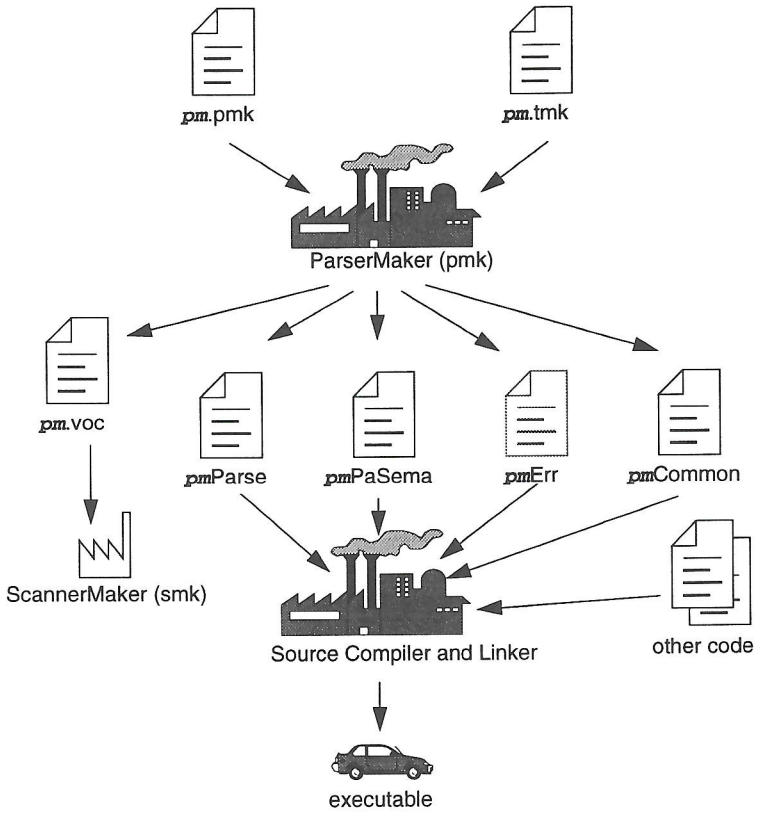


Fig. 1 Files used and produced by ParserMaker.

Depending on the target language used the produced target language modules, may be one or more files, e.g. for 'c' there will be one **pmParse.c** and one **pmParse.h** created for the module **pmParse**.

Following the above figure, this following is a step by step instruction how to use ParserMaker in order to create a working system.

1) Create a ToolMaker description file (***pm*.tmk**) according to the specifications in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. This step is not necessary if you are developing a stand-alone parser (in this case the information can be put inside ***pm*.pmk** instead).

2) Create a ParserMaker description (grammar) file (***pm*.pmk**) according to the specifications in *The ParserMaker Description File* on page 79.

3) Invoke ParserMaker on the grammar file (see *THE PARSERMAKER COMMAND* on page 104). Repeat this step until there are no input errors.

4) Compile the ParserMaker generated components (e.g. ***pm*Parse.c** and ***pm*PaSema.c**).

5) Create and compile other system components including a main program that calls the parser. If you use the ToolMaker kit, a lot of these modules will already have been created for you.

6) Link the object modules and execute.

Points 1 and 2 can be simplified by invoking ***toolmake*** to set up a development environment for you (see the *Toolmake Reference Manual*).

Note: In rare cases the description files are not sufficient to capture all the target dependencies or special requirements for a project, and you have to modify the parser skeletons. If this is the case copy the file **Parse.imp** and **Err.imp** to the development directory from the appropriate library and modify them. TO DO THIS IT IS ABSOLUTELY NECESSARY THAT YOU HAVE A THOROUGH UNDERSTANDING OF THE SYSTEM. Change the library option in the ParserMaker description file to point to the directory where the modified skeletons resides. When ParserMaker is executed, it will use these skeletons instead of the standard ones.

3.1.1 File Descriptions

3.1.1.1 Input Files

ParserMaker Description file (**.pmk**), also known as the G*rammar file*. This file contains a definition of the input grammar according to the specifications in *The ParserMaker Description File* on page 79.

ToolMaker Common Description file (**.tmk**) contains definitions common for all Makers. ParserMaker uses common options and the token definition from this file.

3.1.1.2 Output Files

Table and definition file (**.pmt**). This intermediate file is only kept when the option `Generate Tables` is set. It contains all the definitions and parser tables necessary to create a final parser and is normally removed after successful generation of source files.

Vocabulary file (**.voc**). Contains the representation of the terminal symbols and constitutes the interface between the scanner and the parser. It is used by ScannerMaker to construct a scanner for the language.

List file (**.pml**). The contents of the list file is controlled by the List-directive (see *The List Directive* on page 91). A complete list file contains the following:

- Pretty printed grammar (`List Grammar;`).
- Generated item set (`List Items;`).
- Generated parser tables (`List Tables;`).
- Packed parser tables (`List Tables;`).
- Statistical information (`List Statistics;`).
- Informational messages (`List Info;`).
- Source code listing (`List Input;`).

Parser module (***pm*Parse**). Target language dependent parser module. Created from the **Parse.imp** file where definitions, interfaces and tables have been incorporated. Depending on the target language one or more files may be produced, e.g. for 'c' the files ***pm*Parse.c** and ***pm*Parse.h** are produced. The directives `Pack` and `Recovery` defines much of the layout of the file.

Semantic module (***pm*PaSema**). Target dependent semantic module. The file is created from the skeleton file **Parse.imp** and contains semantic actions and other target dependent code supplied by the parser implementor.

Error recovery interface module (***pm*Err**). Target dependent. The file is created from the skeleton file **Err.imp** and contains error recovery routines that constitute an interface to the ListerMaker list system.

All output files from ParserMaker are created in the current directory.

3.2 The ParserMaker Description File

This section describes the format of the ParserMaker description file. The notation used to do this is a modified EBNF. For a brief description of this refer to appendix C in the *ToolMaker System Description*, *SYNTAX NOTATION*, page 52.

3.2.1 Lexical Definitions

This chapter defines the lexical symbols of the ParserMaker description language.

3.2.1.1 Character Set

Symbols in the ParserMaker language is constructed by using upper case characters (all upper case characters including multi-national ISO-8859 characters), lower case characters (ISO-8859) and digits.

```
<letter> ::=
     <upper case letter> | <lower case letter>
```

All symbols defined by the parser implementor in the ParserMaker description file are case sensitive, e.g. 'Begin' is different from 'BEGIN'. Reserved words and keywords (see below) are not case sensitive.

In the target dependent parts of the ParserMaker language, any character is accepted (see *Common Directives* on page 87 for a description of how to specify the escape character).

3.2.1.2 Lexical Items and Spacing Conventions

The ParserMaker language consists of a sequence of lexical items (tokens). Tokens are identifiers, reserved words, keywords, numbers, strings, delimiters and comments.

The tokens may be separated by any number of spaces, horizontal tabulates or new lines. Tokens are indivisible, space must not occur within tokens, except for strings and comments.

Identifiers are used as names (nonterminals, terminals, attributes), and keywords.

```
<identifier> ::= <letter> {<letter> | <digit> | '_'}
```

All characters in an identifier are significant. Lower case letters are different from upper case letters in all identifiers except keywords. Example:

```
TERMINAL        terminal_definition
```

Reserved words

A reserved word is an identifier preceded by the character pair %%. No space is allowed within the token. The reserved words are predefined and can be used in special contexts only. Letter case is not significant for reserved words. The reserved words are:

```
'%%TERMINALS'
'%%ATTRIBUTES'
'%%RECOVERY'
'%%RULES'
'%%DECLARATIONS'
'%%OPTIONS'
'%%END'
'%%EXPORT'
'%%SCANNER'
'%%INSERTSYMBOL'
'%%IMPORT'
'%%SRCP'
'%%TOKEN'
```

Keywords

A keyword is an identifier with a predefined meaning. A keyword can however be used as any other identifier in the language. Letter case is not significant for keywords. The keywords are

```
'ACTIONPACK'
'ATTRIBUTES'
'CODE'
'COLUMN'
'ERROR'
'ERRORHANDLER'
'ESCAPE'
'FIDUCIAL'
'FILE'
'FORCE'
'GCS'
'GENERATE'
'GOTOPACK'
'GRAMMAR'
'HEIGHT'
'HELP'
'INFO'
'INPUT'
'ITEMS'
'LES'
'LIBRARY'
'LIST'
'LISTERPREFIX'
'LR0'
'LOOKAHEADMAX'
'META'
'MULTIPLE'
'NAME'
'NO'
'OPTIMIZE'
'OS'
```

```
'PACK'
'PANIC'
'POSITION'
'PREFIX'
'RDS'
'RECOVERY'
'RESOLVE'
'ROW'
'RR'
'SEPARATOR'
'SHIFTCOST'
'SINGLE'
'SKIP'
'SR'
'SRCP'
'SOURCE'
'STACKLIMIT'
'STACKSIZE'
'STATISTICS'
'TABLES'
'TARGET'
'TRACE'
'VERBOSE'
'WIDTH'
```

Numbers

A number is an integer literal. It can be written in either decimal or hexadecimal form.

```
<number> ::= <decimal integer> | <hexadecimal integer>

<decimal integer> ::= <digit> {<digit>}

<hexadecimal integer> ::= '#' <hex digit> {<hex digit>}

<hex digit> ::= <digit>
          | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
          | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
```

Example:

```
255       #FF       0         #0
```

Strings

A string in ParserMaker is a sequence of characters, at least one, enclosed in bracket characters. Two string types are used - the quoted string and the angle bracketed string.

```
<string> ::= <quoted string> | <angle bracketed string>

<quoted string> ::=
     ''' <character> {<character>} '''

<angle bracketed string> ::=
     '<' <character> {<character>} '>'
```

If a quoted string is to contain the character '' it must be preceded by the character '\'. The right angle is not allowed in angle bracketed strings.

Example:

```
'*'       '=>'      'BEGIN'   'end'
'\'A quoted string\''

<identifier>       <any character, but %%>
```

Terminals, Nonterminals and Attributes

In the *terminals*, *attributes*, *recovery* and *rules sections* the symbols TERMINAL, NONTERMINAL and ATTRIBUTE are considered terminal symbols. They are defined as follows

```
TERMINAL ::= <identifier>
           | <angle bracketed string>
           | <quoted string>

NONTERMINAL ::= <identifier>
              | <angle bracketed string>

ATTRIBUTE ::= <identifier>
```

Delimiters and Special Tokens

ParserMaker defines the following single character delimiters:

```
'='
';'
','
'('
')'
'!'
'{'
'}'
'|'
'['
']'
```

and the special tokens:

```
'%%'
'%+'
'%-'
'=>'
```

Comments

A comment starts with two dashes, '--', and is terminated by an end of line. A comment has no effect on the meaning of the ParserMaker language.

Example:

```
        -- A single comment

    -- A long comment
    -- must be split into several lines
```

Escape character

In the semantic actions, the character '‘' is used as an escape character. A character following the escape character will always be copied to the output. The escape character may be changed by means of the escape option (see *Common Directives* on page 87).

Example: to include the character '%' in a semantic action it must be written '‘%'.

> *Note:* The escape character is valid only in the target dependent code parts of the input description.

3.2.2 Overall Structure of the Description File

```
<description file> ::=
     <toolmaker sections>
     <other sections>
     <rules section>

<toolmaker sections> ::=
     [ <options section> ]
     { <import section> | <srcp section>
          | <token section> }

<other sections> ::=
     { <declarations section>
     | <terminals section>
     | <attributes section>
     | <recovery section>
     | <export section>
     | <scanner section>
     | <insertsymbol section>
     | <deletesymbol section> }
```

The ParserMaker description file may contain thirteen different sections: the *options section*, the *import section*, the *srcp section*, the *token section*, the *declarations section*, the *terminal section*, the *attribute section*, the *recovery section*, the *export section*, the *scanner section*, the *insertsymbol section*, the *deletesymbol section* and the *rules section*.

All sections except the *rules section* are optional. The rules section must be the last section, the option section must be first if it exists and the import, token and srcp sections must, if they exist, immediately follow the option section. Otherwise the order between the sections is free.

The *options section* controls the ParserMaker generation process. For instance, the degree of table packing, included error recovery facilities and the kind of logging information produced.

The *import, token* and *srcp sections* are further described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. These three sections are normally located in the ToolMaker description file and should be put in the ParserMaker description file only when developing a stand-alone parser.

The *declarations, export, insertsymbol, deletesymbol* and *scanner sections* defines target dependent code that should be included in the produced parser.

The *terminals*, *attributes* and *recovery sections* together with the *rules section* specifies the input language. It contain facilities for tuning the automatically generated error recovery system, defining the scanner interface, handling ambiguous grammars and defining semantic actions.

3.2.3 An Example

To give a short overview of what the description file may look like, a small example will be presented.

The problem: Assume that a file contains a sequence of telegrams.

Each telegram consists of one or more sentences terminated by the symbol 'STOP'. A telegram ends with the symbol 'ZZZZ'. The problem is to analyse a telegram file and for each telegram write a summary containing the total number of words in the telegram (excluding STOP and ZZZZ) and all words exceeding 12 characters.

Preconditions: The target language is 'c'. A scanner exists which recognizes the tokens:

```
WORD
'STOP'
'ZZZZ'
END-OF-FILE
```

When a WORD is recognized the scanner returns the length of the WORD. The lexical attribute LENGTH is used to hold this value. The scanner call interface is defined in the scanner section (see *The Scanner Section* on page 96). The token section is shown here for completeness but is normally defined in the ToolMaker description file.

Example: A file containing the text:

```
I WILL ARRIVE AT 5 PM STOP THE HOTEL IS
INTERNATIONALE STOP
ZZZZ
```

will result in the output:

```
Words = 10
Words > 12 = 1
```

The solution:

```
%%TOKEN
     CODE code;
     ATTRIBUTES
          LENGTH;

%%SCANNER tgrScan(tgrCtxt, token);

%%DECLARATIONS

#include <stdio.h>
int wordCount;
int moreThan12Chars;

%%TERMINALS
     WORD = 2;
     'STOP' = 3;
     'ZZZZ' = 4;

%%RULES

telegrams
     = telegram
     ! telegrams telegram
     ;

telegram
     =
       %%
          wordCount = 0;
          moreThan12Chars = 0;
       %%
     sentences 'ZZZZ'
       %%
          printf("Words = &%u\n", wordCount);
          printf("Words > 12 = &%u\n", moreThan12Chars);
       %%
     ;

sentences
     = sentence
     ! sentences sentence
     ;

sentence
     = word_list 'STOP'
     ;

word_list
     = element
```

```
        ! word_list element
        ;

element
        = WORD
          %%
             wordCount++;
             /* Look at the length */
             if (%WORD.length > 12) {
                  moreThan12Chars++;
             }/*if*/
          %%
        ;
```

3.2.4 The Options Section

```
<options section> ::=
     '%%OPTIONS' <directive> {<directive>} ['%%END']

<directive> ::=
       <common directive>
     | <listerprefix directive>
     | <errorhandler directive>
     | <trace directive>
     | <lookaheadmax directive>
     | <shiftcost directive>
     | <stacklimit directive>
     | <pack directive>
     | <line directive>
     | <list directive>
     | <optimize directive>
     | <recovery directive>
     | <resolve directive>
     | <vocabulary directive>
     | <table file directive>
     | <list file directive>
```

The *options section* of the ParserMaker description file controls the parser generation process. Corresponding options can be given as command line parameters to ParserMaker at its invocation. Options given as command line parameters override those within the description file. Command line options are described in *Options* on page 104. For a detailed discussion on options see *The Options Section* in the *ToolMaker System Description*, page 30.

The directives can be given in any order. If a directive is repeated, only the last one given is used.

Note that if a directive is explicitly stated in the *options section*, all its default options are turned off. This means that all desired options must be included when a directive is used explicitly, not just the desired extra ones, which is especially important for set valued directives. The default setting of the options are

```
Listerprefix 'tm';
Errorhandler;
No Trace;
Lookaheadmax 5;
```

```
Shiftcost 5;
Stacklimit 32;
Pack RDS;
No Line;
List Info;
Optimize Lr0;
Recovery Single, Multiple, Panic;
Resolve SR;
```

3.2.4.1 Common Directives

```
<common directive> ::=
        <target directive>
      | <os directive>
      | <prefix directive>
      | <library directive>
      | <escape directive>
      | <width directive>
      | <height directive>
      | <generate directive>
      | <force directive>
```

The common directives are directives available for all of the Makers in the ToolMaker kit. For a detailed description of these refer to *The Options Section* in the *ToolMaker System Description*, page 30. All directives are available for ParserMaker, and if used overrides default values and settings in the ToolMaker Common Description file.

The prefix directive does not inherit its default value, instead it defaults to '**pm**' if not explicitly set in the **.tmk** file. If set in the ToolMaker Common Description file and *not* used in the ParserMaker Description file it defaults to the system prefix (the value set in the ToolMaker Common Description file).

3.2.4.2 The ListerPrefix Directive

```
<listerprefix directive> ::=
     'LISTERPREFIX' <quoted string> ';'
```

To be able to interface the generated error message handler to ToolMaker generated Lister modules (see *ListerMaker Reference Manual*) that does not use the default prefix this directive can be used. The quoted string will be used as a prefix in all function calls to the Lister.

The default is the system prefix, i.e. ParserMaker assumes that the Lister is generated using the system prefix, which unless set otherwise in the ToolMaker description file is 'tm'.

3.2.4.3 The Errorhandler Directive

```
<errorhandler directive> ::=
     ['NO'] 'ERRORHANDLER' ';'
```

This option turns on [off] generation of the parser error handling interface routines. See *Run-Time Interface* on page 107 for a definition of these routines. The default is:

```
ErrorHandler;
```

3.2.4.4 The Trace Directive

```
<trace directive> ::=
     ['NO'] 'TRACE' ';'
```

The trace directive instructs ParserMaker to include *trace printing* functions in the generated parser. This information can, together with the information in the list file give information which simplifies the debugging of the grammar.

The default value of the trace directive is:

```
No Trace;
```

3.2.4.5 The LookAheadMax Directive

```
<lookaheadmax directive> ::=
     'LOOKAHEADMAX' <number> ';'
```

This option defines *the maximum amount of look-ahead that will be applied when the parser detects a syntax error.* The default value of 5 represents a reasonable functionality versus performance trade-off. See *Error Recovery Principles* on page 72 for a discussion of the impact of this option. Any positive integer value larger than or equal to 4 may be used as value. Due to performance reasons, it is not recommended to increase the look-ahead above 6.

3.2.4.6 The ShiftCost Directive

```
<shiftcost directive> ::=
     'SHIFTCOST' <number> ';'
```

This option defines *the acceptance cost for shifting one symbol from the input.* See *Error Recovery Principles* on page 72 for a discussion of the impact of this option. Any positive integer may be used as the value. The default is:

```
ShiftCost 5;
```

3.2.4.7 The StackLimit Directive

```
<stacklimit directive> ::=
     'STACKLIMIT' <number> ';'
```

This option defines *the size of the parse stack*, and the default value of 32 should be enough for all but the most demanding situations. If a parse stack overflow occurs in the generated parser (indicated by error message 106 in the

default error handler), this constant can be increased. Note however that the most common reason for parse stack overflow is that the grammar is *right recursive*, i.e. it contains one or more rules of the form:

```
<a>  = b <a>
     ! b
     ;
```

where b may be one or more symbols. This is not an error per se, but writing rules this way prevents any reductions to be performed until an entire construct, e.g. an entire list, has been recognized. If the text to be analysed contains long lists, the parse stack may grow out of bounds. The most general way to deal with this kind of problem is to transform the grammar to be *left recursive*, i.e. to eliminate all right recursive rules. The rule above can be transformed to:

```
<a>  = <a> b
     ! b
     ;
```

This form of rule permits a reduction to be performed for every element of the construct, e.g. for every element of a list, and parse stack size will normally not be a problem. Note however that care must be taken when the rules are transformed so that an equivalent grammar is constructed.

If parser size is an issue, this constant should be decreased since each stack entry will occupy memory needed to accommodate all of the defined attributes, both the terminal attributes and the non-terminal attributes. The stack size may be set to any positive integer value.

The default value is:

```
StackLimit 32;
```

3.2.4.8 The Pack Directives

```
<pack directive> ::=
       'PACK' <pack option>
              {',' <pack option>} ';'
     | ['NO'] 'PACK' ';'
     | 'ACTIONPACK' <pack option>
              {',' <pack option>} ';'
     | 'GOTOPACK' <pack option>
              {',' <pack option>} ';'

<pack option> ::=
     'ROW' | 'COLUMN' | 'RDS' |
     'GCS' | 'LES' | 'ERROR'
```

The pack directives guides the parse table packing. The directive PACK will pack both action and goto tables, ACTIONPACK will only affect the packing of the action table and finally GOTOPACK will only affect the packing of the

goto table. High degree of table packing normally increases the execution time.

The implication of the ROW and COLUMN options is that identical rows and columns of the tables will be merged.

The RDS option stands for *Row Displacement Scheme*. With this method the table will be linearized to a vector. The rows of the original table will overlap each other in the constructed vector, and an attempt is made to store only significant elements. If used in conjunction with GCS or LES, the error matrix method will be used for error detection, otherwise a check vector will be constructed in order to detect error entries.

The GCS option stands for *Graph Colouring Scheme*. This means that compatible rows and columns will be represented by a single row or column. When this packing technique is used, a binary error matrix will be constructed for error detection.

The LES option stands for *Line Elimination Scheme*. This means that columns and rows with a single action will be moved out from the table. A binary error matrix is constructed for error detection.

The implication of the ERROR option is that identical rows and columns of the error matrix will be merged. If the contents of the unpacked error matrix is not unfavourable, and the space requirements for the table is not substantial, the effect of this packing method may in fact result in increased space requirements!

If NO PACK is specified, there will be no packing of the tables.

If you specify PACK without any options or if this directive is not specified, the default setting is:

```
Pack RDS;
```

which gives a reasonable trade-off between table space and execution time. Refer to for more information on how to select which table packing to use.

3.2.4.9 The Line Directive

```
<list directive> ::= ['NO'] 'LINE' ';'
```

The line directive instructs ParserMaker to generate line number information in the generated source so that compilers and debuggers will belive that the source is the actual description file instead of the generated source file. This can not be handy in initial stages of compiling the semantic actions and debugging, but as advanced debugger allows you to point in the source to find variables it is not always good. Also this option might not have any effect for

every of the supported languages (cf. the 'c'-language preprocessor directive `#line`).

3.2.4.10 The List Directive

```
<list directive> ::=
       'LIST' <list option>
              {',' <list option>} ';'
     | ['NO'] 'LIST' ';'

<list option> ::=
       'INPUT' | 'GRAMMAR' | 'ITEMS'
     | 'TABLES' | 'STATISTICS' | 'INFO'
```

The list directive controls the output listings from the parser generation.

With the `INPUT` option activated, a listing of the input description file with the input grammar and error messages concerning the input will be placed in the *list file* (**.pml**).

The `GRAMMAR` option activates the output of a *formatted version of the input grammar* to the list file.

Options `ITEMS` and `TABLES` output the *generated item set* and *the produced tables*. For large grammars the output listings becomes huge if these options are set. Even if the options are reset, error messages and appropriate parts of the item set will be produced in case of errors in the generation process.

The `STATISTICS` option indicates that some *statistical information* will be printed on the list file.

The `INFO` option indicates that *informational messages* (or more precisely, messages with information severity) will be printed to the list file and the terminal. The implication of having the `INFO` option turned off is that only error messages with warning severity or greater will be printed. This can be useful if a huge amount of uninteresting informational messages is produced, thus preventing interesting error messages to be shown.

If the List directive is specified as `LIST` (without any options) or not specified at all, the default options are:

```
List Info;
```

3.2.4.11 The Optimize Directive

```
<optimize directive> ::=
       'OPTIMIZE' [<optimize option>
                  {',' <optimize option>}] ';'
     | ['NO'] 'OPTIMIZE' ';'

<optimize option> ::= 'LR0'
```

The optimize directive guides the level of optimization applied when the parse tables are computed. Currently there is only one option, LR0, which will remove a certain kind of parser states - *LR(0) reduce states*. This can significantly reduce the size of the parser tables.

If NO OPTIMIZE is specified there will be no optimization.

If OPTIMIZE is specified without options or if the directive is left out, the default setting is:

```
Optimize Lr0;
```

3.2.4.12 The Recovery Directive

```
<recovery directive> ::=
       'RECOVERY' [<error option>
                   {',' <error option>}] ';'
     | ['NO'] 'RECOVERY' ';'

<error option> ::=
       'SINGLE' | 'MULTIPLE' | 'PANIC'
```

The recovery directive defines the degree of error recovery to apply.

Option SINGLE implies that a first level of *single symbol error correction* should be applied.

Activating the MULTIPLE option means that the *second level* should be invoked when the first fails. If SINGLE is not activated, the second level is activated first. The second level correction continues until the string is corrected or until it tries to delete an important symbol (fiducial) and the PANIC option is specified.

PANIC indicates that the last step in the recovery is pure *panic mode*. Refer to the section *Error Recovery Principles* on page 72 for further details.

NO RECOVERY means that the parser will *abort the parsing process* if an error is detected.

If the recovery directive is specified as RECOVERY (without any options) or not specified at all, the default options are:

```
Recovery Single, Multiple, Panic;
```

3.2.4.13 The Resolve Directive

```
<resolve directive> ::=
       'RESOLVE' <resolve option>
                   {',' <resolve option>} ';'
     | ['NO'] 'RESOLVE' ';'
```

```
<resolve option> ::=
       'SR' | 'RR'
```

The resolve directive defines how to react to ambiguities in the input grammar.

Option SR implies that a shift-reduce conflict shall be resolved by using shift in favour of reduce.

Option RR implies that a reduce-reduce conflict shall be resolved by reducing the production that comes first in the input grammar.

If `NO RESOLVE` is specified *no tables are created* if conflicts are not removed by modifications, i.e. the generation is aborted on any conflict. Also refer to *Grammar Ambiguity and LALR-conflicts* on page 68.

The default setting, which you get if the directive is not specified or if it is specified without options, is:

```
Resolve SR;
```

3.2.5 The Import, Export and Declarations Sections

```
<import section> ::=
     '%%IMPORT' <target language code> ['%%END']

<export section> ::=
     '%%EXPORT' <target language code> ['%%END']

<declarations section> ::=
     '%%DECLARATIONS' <target language code> ['%%END']
```

The semantic actions in the *rules section* are written in the target language extended with an attribute propagation technique. Variables and subroutines referenced within the semantic routines are defined in the *import* and *declaration sections*. Using the *export section* variables and functions declared in the declaration section may be exported. These particular sections should contain *declarations and routines in the target language*. The character sequences, representing the code, are copied unformatted to the output file.

> *Note:* If the character sequence '%%' appears in the code, at least one of the percent signs (preferably both) must be quoted with the escape character.

Example: A 'c' interface might be written:

```
%%EXPORT

extern void push(element);
extern int pop();
```

```
%%DECLARATION

#define LENGTH 10
static int stack[LENGTH];
static int ptr = 0;

void push(element)
int element;
{
     if (ptr < LENGTH) stack[ptr++] = element;
}/*push()*/

int pop()
{
     if (ptr > 0) return stack[--ptr];
}/*pop()*/
```

Target dependent errors in the *import, export and declaration sections* will not be detected until the generated code is run through the target compiler.

3.2.6 The Terminals Section

```
<terminals section> ::=
     '%%TERMINALS' {<terminal definition> ';'} ['%%END']

<terminal definition> ::=
     TERMINAL '=' <token code> [<error recovery data>]

<token code> ::= <number>

<error recovery data> ::=
     ',' <insert cost> ',' <delete cost>[<print symbol>]

<insert cost> ::= <number>

<delete cost> ::= <number>

<print symbol> ::= '=>' TERMINAL
```

The *terminals section* defines the scanner interface and is optional. If omitted, ParserMaker will supply pertinent defaults. If present, it should define the terminal symbols used in the grammar and the token codes returned by the scanner. Information for error recovery improvement can be supplied with each terminal.

The left hand side of the definition is a terminal symbol. The terminal is written exactly as it is used in the subsequent grammar. The `<token code>` is a positive integer value that the scanner must return in order for the generated parser to recognize the token. If the *terminals section* is omitted ParserMaker will generate an internal number for each token.

Example:

```
%%TERMINALS
     <identifier> = 2;
```

```
NUMBER = 3;
';' = 4;
```

Note: Code values 0 and 1 must not be used. Token code 0 is reserved by the system and code 1 is used for the endmarker. The endmarker must not be included in the terminal list.

The optional error recovery information makes it possible for a parser implementor to tune the error recovery handler. The insert cost and delete cost values define the cost of inserting and deleting the terminal symbol respectively. The higher the cost, the more seldom the symbol will be inserted or deleted in the error recovery process. Both the insert cost and the delete cost default to one (1).

The print symbol defines the string to be used in error messages and in the error repair process instead of the terminal symbol itself. This facility applies normally only to pseudo terminals, i.e. terminals like identifiers and numbers, which are identified both with a type and a value. The default print symbol is the terminal string itself.

Example: An identifier might be defined as:

```
%%TERMINALS
     <identifier> = 10, 5, 1 => 'GENID00';
```

An identifier has the token code `10`, and it is expensive to insert an identifier compared to deleting it (`5` versus `1`), and if an identifier is inserted, `'GENID00'` will be issued in its place in the error message.

Example: The definition:

```
<identifier> = 10;
```

is equivalent to:

```
<identifier> = 10, 1, 1 => '<identifier>';
```

How these weights and the print name are used is described in *Error Recovery Principles* on page 72, in *The Insertsymbol Section* on page 97, *The Deletesymbol Section* on page 97 and in *Error Recovery* on page 106.

3.2.7 The Attributes Section

```
<attributes section> ::=
     '%%ATTRIBUTES'
     <grammar attribute> {',' <grammar attribute>} ';'
     ['%%END']

<grammar attribute> ::=
     <identifier> [ '%%' <target language code> '%%' ]
```

The *attributes section* defines the semantic attributes used in the grammar. Language dependent declarations, i.e. a description of the implementation, are defined in the optional declaration part of each attribute.

Example:

```
%%ATTRIBUTES
     ival %% int ival %%,
     pos  %% long pos %%;
```

Instead of repeating the name of the identifier in the target language declaration, it is possible to use a '%1' as a placeholder for the name. It will automatically be substituted with the name in the produced code.

Example:

```
%%ATTRIBUTES
     ival %% int %1 %%,
     pos  %% long %1 %%;
```

How to use attributes is described in *Grammar Attributes* on page 64.

3.2.8 The Scanner Section

```
<scanner section> ::=
     '%%SCANNER' <target language code> ['%%END']
```

The code specified in the scanner section is executed when a new token should be read from the input stream. The target language code is copied into the generated parser. The purpose of this code is to fill the token structure as supplied by the pre-defined pointer variable `token` which is defined when the scanner section is entered. See *The Token Section* in the *ToolMaker System Description*, page 36 for a description of the fields in the token structure and how to define them.

The *scanner section* can be viewed as a procedure with the following formal interface:

scanner section`(lexContext, token)`

```
lexContext: INOUT %%(ScannerContext)
token: OUT %%(tokenType)
```

Example:

```
%%IMPORT
#include "smScan.h"

%%SCANNER
     smScan(lexContext, token);
%%END
```

If this section is omitted, default code will be generated:

```
%%(tmkPrefix)Scan(lexContext, token);
```

where `%%(tmkPrefix)` stands for (and will be expanded to) the system prefix, i.e. the code generated will assume that the scanner is a scanner generated by ScannerMaker using the common prefix (see *The Prefix Directive* in the *ToolMaker System Description*, page 32 for a description of how to define a system prefix).

3.2.9 The Insertsymbol Section

```
<insertsymbol section> ::=
     '%%INSERTSYMBOL' <target language code> ['%%END']
```

This block should contain the necessary *target language code to construct a semantically valid token* when the parser error recovery process has decided to insert a token in the input stream. The code is inserted into the **pm**`ISym()` function. See *Run-Time Interface* on page 107 for further details.

3.2.10 The Deletesymbol Section

```
<deletesymbol section> ::=
     '%%DELETESYMBOL' <target language code> ['%%END']
```

This block should contain the necessary *target language code to handle the deletion of an already constructed token* when the parser error recovery process has decided to delete a token from the input stream. The code is inserted into the **pm**`DSym()` function. See *Run-Time Interface* on page 107 for further details.

3.2.11 The Recovery Section

```
<recovery section> ::=
     '%%RECOVERY'
     [<meta part>]
     [<separator part>]
     [<fiducial part>]
     [<skip part>]
     ['%%END']
```

The *recovery section* is optional. It defines language dependent information that will improve the error recovery system.

3.2.11.1 The Meta Part

```
<meta part> ::=
     'META' {<meta name> '=' '(' TERMINAL
          {',' TERMINAL} ')' ['=>' TERMINAL] ';'}
```

The meta part defines alias names for groups of symbols. For example, OPERATOR can be defined to be the alias name for all operators. The alias name

will then be used both in error messages and in the first level repair process. This facility will improve the quality of error messages and the speed of the recovery process.

The `meta name` is the name of the alias. The terminal list defines the symbols that is covered by the alias name. The terminals within the list must have equal insertion costs. Any specific terminal must not appear in more than one meta definition. The meta name will be used only when all the terminals in the list are accepted in the error state. Only the single symbol error correction is affected. The optional `'=>' TERMINAL` means that the indicated terminal will be used in the repair process. If not specified, an arbitrary symbol is used.

Example:

```
META 'UNARY' = ('-', '+', 'NOT') => '-';
```

means that `UNARY` will be used in error messages for errors concerning -, + and `NOT`. If a unary operator is missing, the symbol - will be used in the correction.

3.2.11.2 The Separator Part

```
<separator part> ::=
     'SEPARATOR' '(' TERMINAL {',' TERMINAL} ')' ';'
```

The separator part is used to improve the second level recovery. It contains symbols which separates recursive lists in the grammar. Specifying a set of separators will result in greatly improved second level recovery, since the number of grammar productions that are taken into account during error recovery will become significantly larger.

Example: for a Pascal grammar, the following declaration would be appropriate:

```
SEPARATOR(',', ';', ':');
```

3.2.11.3 The Fiducial Part

```
<fiducial part> ::=
     'FIDUCIAL' '(' TERMINAL {',' TERMINAL} ')' ';'
```

The fiducial part defines the important symbols for the second level and panic mode recovery techniques. *Fiducial symbols* are symbols that start significant structures of the specified language.

For Pascal, `CONST`, `VAR`, `TYPE`, `LABEL`, `PROCEDURE` and the header symbols of statements, such as `BEGIN`, `WHILE`, `FOR`, could be selected.

3.2.11.4 The Skip Part

```
<skip part> ::=
     'SKIP' '(' TERMINAL {',' TERMINAL} ')' ';'
```

The skip part is intended to improve the second level recovery. The symbols specified in the skip part has the opposite meaning to the symbols in the fiducial part. Skip symbols are tokens that appear in many contexts and should not be regarded as safe restarting symbols.

The specified symbols will always be deleted in the second level error recovery process. In Pascal, identifiers and constants are good skip symbol candidates.

3.2.12 The Rules Section

```
<rules section> ::=
     '%%RULES' {NONTERMINAL '=' <alternatives> ';'}
     ['%%END']

<alternatives> ::=
     <right hand side> {'!' <right hand side>}

<right hand side> ::= <components> <opt modify>

<components> ::= {<component>}

<component> ::=
       <symbol>
     | <action modify>
     | '(' <components> {'|' <components>} ')'
     | '{' <components> '}'
     | '[' <components> ']'

<symbol> ::= TERMINAL | NONTERMINAL

<action modify> ::=
     <opt modify> <semantic action> <opt modify>

<semantic action> ::=
     '%%' <any character sequence, but %%> '%%'

<opt modify> ::=
     {('%+' | '%-') '(' TERMINAL {',' TERMINAL} ')'}
```

The *rules section* defines the syntax of the input language. The input specification is written in an EBNF-like notation, extended with semantic actions and a mechanism for resolving ambiguities in the grammar.

The concepts behind the various constructs in the *rules section* are discussed at length in *CONCEPTS AND ASSUMPTIONS* on page 60.

3.3 The ToolMaker Common Description File

Unless ParserMaker is the only Maker used, common declarations of the source position and the token structures should be placed in the ToolMaker Common Description file which is described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. Otherwise these two sections may be specified in the ParserMaker Description file, removing any need for the ToolMaker Common Description file.

3.4 The List File

The list file is used for various purposes:

- list the input and point out syntactic and other errors.
- log information about the parser generation process.
- log information about errors encountered in the process.

This chapter contains information on how to interpret the contents of the list file.

3.4.1 Format of the Pretty Printed Grammar

The pretty printed grammar is written to the list file if option *List Grammar*; is activated. It is divided in two parts, the *vocabulary* and the *productions*.

The *vocabulary* contains:

- The *terminals* with their internal code, token code, insertion cost, deletion cost and print symbol.
- The *non-terminals* with their internal code.

The *productions* list contains all productions pretty printed together with their *production number*.

The internal codes of the vocabulary and the production numbers are later used when the item set and the tables are printed.

3.4.2 Format of the Generated Item Set

The LALR(1) item set is written to the list file if the option `List Items;` is activated. A subset of the item set is also written if the grammar contains errors. This may provide useful information in conjunction with the trace output printed if the trace directive is set in the options section (see *The Trace Directive* on page 88). The concept of items is further discussed in reference [Aho].

For each state, a table containing all items in that state is written. The table is labelled by the *state number* and the *continuation symbol* chosen by the second level error correction (see *Level 2: String Synthesizing Technique* on page 74). If *separators* have been defined (see *The Separator Part* on page 98), a *separator continuation symbol* will be printed within parentheses if applicable for the state.

Each item in the state is represented by an entry in the table. Each entry is labelled by the *parsing action* for the item in the state. The action has one of the following formats:

- `<number>`. The action for the item is *shift* if the symbol following the dot is recognized. The *number* indicates the next state.
- `'R'<number>`. The action for the item is *reduce* by production *number*. The production number refers to the number in the productions list.
- `'SR'<number>`. The action for the item is *shift-reduce* by production *number*.

The item itself follows the parsing action. If the parsing action is *reduce*, the item is followed by the *follow set* of terminals within curly braces.

Consider the following example:

```
State:   14   !  Continuation: '.'(';')
- - - - - - - ! - - - - - - - - - - - - - - - - - - - - -
          R39 ! <terms> --> <terms> <addop> <term> .
              !       {'.' ';' '=' '<' '>' '<>' '<=' '>='
              !            ')' '+' '-' 'END' 'THEN' 'DO' }
           15 ! <term> --> <term> .<mulop> <factor>
         SR50 ! <mulop> --> .'*'
         SR51 ! <mulop> --> .'/'
```

The state number is 14, the continuation symbol is '.', and the separator continuation symbol is ';'. The parsing action for the first item is reduce by production 39, and the follow set which causes that reduction is enclosed in curly braces. The parsing action for the second item is shift, and the next state is number 15 if a `<mulop>` is recognized. The parsing action for the third item is shift-reduce by production 50 if a '*' is the next token.

When the grammar is *ambiguous*, indicated by error message 303, the list file will contain information about the ambiguity. If this is the case, the item sets for all offending states will be logged, and each state will be preceded by a line explaining the nature of the ambiguity (shift-reduce or reduce-reduce), followed by the offending symbol and the involved production or productions. With this information it is normally possible to rewrite the grammar or use modification rules to resolve the ambiguity (see *Grammar Ambiguity and LALR-conflicts* on page 68).

3.4.3 Format of the Parser Tables

The parser tables are printed as matrices. Each *row* is labelled with a *state number*, and the state is also identified by the symbol before the dot in the first item of the state. The *columns of the action table* are labelled with the internal codes of the *terminals*, and the *columns of the goto table* are labelled with the internal codes of the *non-terminals*. An entry in the table has one of the following formats:

- `'S'<number>` meaning *shift* the symbol and goto state *number*.
- `'R'<number>` meaning *reduce* using production *number*.
- `'-'<number>` meaning *shift* the symbol and *reduce* using production *number*.
- `'A'` meaning *accept* the input.
- `'X'` meaning *error*.
- `'*'` meaning *don't care*, i.e. the entry can never be reached.

If table packing is activated, the packed parser tables and the various access structures will also be printed. In order to understand the purpose of the various access structures it is necessary to understand the theory behind the various packing methods, see reference [Sencker]. The entries in the packed parser tables have the same format as described above.

3.5 The Vocabulary File

The vocabulary file contains the representation and codes of the terminal symbols, and it constitutes the interface between the scanner and the parser. If ScannerMaker is used, it uses this file to construct a scanner for the vocabulary of the language.

The vocabulary file contains one line for each terminal that the parser expects the scanner to recognize. Each line has the following format:

```
<vocabulary entry> :==
     <sequence number>
     <scanner code>
     <terminal string>
     <vocabulary name>

<sequence number> ::= <decimal integer>

<scanner code> ::= <decimal integer>

<terminal string> ::= TERMINAL

<vocabulary name> :== 'main'
```

The `<sequence number>` is a pure sequence number starting from 1.

The `<scanner code>` is the number the scanner is supposed to return to the parser in the code member of the token structure (see *The Token Section* on page 36) when the corresponding token has been recognized.

The `<terminal string>` is the string representing the terminal to be recognized. The appearance is exactly the same as in the grammar file. This may be either a string or an identifier representing lexical items needing further definition (e.g. by regular expressions in a scanner generator).

> *Note:* Only normal identifiers can be used in the vocabulary file, i.e. as identifiers for tokens. Angle bracketed strings are *not* recognized by ScannerMaker.

Last on each line is an identifier indicating to which vocabulary this token belongs. This is to comply with the ScannerMaker ability to handle multiple vocabularies (see *Vocabulary* on page 134 and *The Vocabulary file* in the *ScannerMaker Reference Manual*, page 160).

> *Note:* ParserMaker assumes that all tokens should belong to the main vocabulary, but by editing this file (e.g. automatically through a stream editor) tokens may be placed in different vocabularies.

4 THE PARSERMAKER COMMAND

To invoke ParserMaker the following command should be given:

```
pmk [-help] [options] <input grammar>
```

The special option `-help` will cause ParserMaker to print a short description of the parameters and available options and then immediately exit.

4.1 Parameters

ParserMaker takes only one parameter, `<input grammar>`, which is the name of the ParserMaker, or grammar, description file. If no extension is given, **.pmk** is assumed.

4.2 Options

Any number of options may be given on the command line. These options override or complement the options given in the *options section* in the grammar file. The Command Line Options and their corresponding directives in the *options section* are the following:

```
Usage: pmk [-help] [options] <input file>
Options:
  -[-]verbose          enable[disable] verbose mode
  -target <lang>       generate file for target language
                           <lang>
  -os <os>             generate source files for target
                           operating system <os>
  -[-]prefix [<prefix>] set [no] parser prefix
  -library <lib>       use directory <lib> for library files
  -[-]escape [<c>]     set [no] escape character
  -[-]width [<n>]      set [no] listing width
  -[-]height [<n>]     set [no] listing height
  -[-]generate         select [no] generated output
                           {tables | source}
  -[-]force        do [not] force generating of source code
  -[-]listerprefix [<prefix>] set [no] lister prefix
  -[-]errorhandler     enable [disable] generation of error
                           handler
  -[-]trace            enable [disable] trace mode
  -lookaheadmax <n>    set max lookaheadto <n>
  -shiftcost <n>       set shift cost for terminals to <n>
  -stacklimit <n>      set parse stack limit to <n> entries
  -[-]pack             set [no] table packing { row | column
                           | rds | gcs | les }
  -[-]actionpack       set [no] packing of action tables
  -[-]gotopack         set [no] packing of goto tables
  -[-]list             set [no] listings { input | grammar |
                        items | tables | statistics | info }
  -[-]optimize         set [no] optimize mode
  -[-]recovery         set [no] recovery mode { single
                           | multiple | panic }
  -[-]resolve          set [no] resolve mode
  -voc <file>          write vocabulary to <file>
```

```
-pml <file>        write lists to <file> (if any)
-pmt <file>        write tables to <file> (if any)
-tmk <file>        read common options from <file>
-help              this help information
```

Refer to the corresponding directives and options in *The Options Section* on page 86 for a detailed description of the options.

The `-voc`, `-pmt` and `-pml` options are only available from the command line and are used to give explicit names to the *vocabulary file*, the *table and definition file* and the *list file* respectively (see *Output Files* on page 78).

4.3 Example

The command line:

```
pmk --pack -generate tables -list items pl0
```

tells ParserMaker to take the file **pl0.pmk** as the input grammar description, not to use any table packing, to list the generated item sets in the list file and to just generate the table file (no source files).

5 PARSER RUN-TIME USAGE

In the descriptions below, all functions external to the generated parser are described with their default prefix pm. This prefix may be changed with the parser prefix option (see *The ToolMaker Description File* in the *ToolMaker System Description*, page 27).

5.1 Principles of Operation

5.1.1 Parsing

When the parser implementor wants to parse the input stream, all that need be done is to call the ***pm***`Parse()` function. The only requirement the parser has is that the input stream must be initialized so that the first call of the scanning function defined in the scanner section of the description file will return the first symbol of the input stream to be parsed. If ScannerMaker is used to generate the scanner, this involves initialising it and setting up a context. Of course it is appropriate to perform other initialization like initializing the listing system, setting data structures for the semantic actions etcetera before calling ***pm***`Parse()`.

5.1.2 Scanning

The generated parser expects the code defined in the scanner section (or default) to supply it with tokens from the input stream. This functionality must be implemented by the parser implementor. If ScannerMaker is used this is no problem since this is exactly the task of ScannerMaker.

The parser expects the scanner section to return exactly one token from the input stream each time it is called. The tokens it expects the scanner to recognize are defined together with their token codes in the vocabulary file (see *The Vocabulary File* on page 102 for the format of the vocabulary file).

5.1.3 Error Recovery

The error recovery system of the generated parser is very good at recovering from syntax errors. What is lacking is a means for reporting to the user what has occurred during error recovery. The *error recovery interface routines* of the generated parser provide the necessary hooks for reporting the error recovery activities to the user. The error recovery interface routines are ***pm***`ISym()`, ***pm***`DSym()`, ***pm***`Mess()` and ***pm***`RPoi()`. These routines can be automatically generated if the `errorhandler` option is used. The default interface assumes a ListerMaker generated lister module. The error recovery interface may also be completely provided by the implementor.

When an error is detected by the parser, the generated error recovery system in the parser is activated to try to repair the error. If this is successful, and if recovery has been made by inserting or deleting one or more symbols,

pmISym() and/or **pm**DSym() are called one or more times (once for each inserted or deleted symbol).

> *Note:* when the error recovery method is *replacement* of symbols, *both* of these routines will be called.

Regardless whether the recovery process was successful or not, the next step is that **pm**Mess() is called to report that the error recovery process is finished, and the result of the process. At this time it is appropriate to output an error message to the poor parser user.

If the error recovery process was successful, i.e. the parser was able to recover from the error, **pm**RPoi() is called to indicate from what point the parser will continue to process the input (the recovery point). For small errors, this is often the same point that caused the error, and in these cases it is probably not necessary to do any special reporting to the user.

In summary, this is the calling sequence of the error recovery interface routines:

1) **pm**ISym() and/or **pm**DSym() are called zero or more times depending on the recovery method selected.

2) **pm**Mess() is called once.

3) **pm**RPoi() is called once if the error recovery process was successful.

After these corrections of the input has been made the parsing is continued as if the input had actually been the corrected stream of tokens. No semantic actions are executed during error recovery until the final decision on which corrections to apply has been made, and then it is executed exactly as if the corrected input had been found. This means if the routines above (particularly **pm**ISym() as defined by the *insertsymbol section*) have been designed properly the semantic actions will be executed exactly as if the user had input the corrected input, thus making it possible to perform even semantic analysis on the input. The design of the semantic actions is thus greatly simplified as no special care have to be taken for the erroneous input.

5.2 Run-Time Interface

5.2.1 Function: **pm**Parse()

```
pmParse()
```

The parser entry point. To be called from the application when a source language text should be parsed. The scanner must be initialized when this function is called so that the *scanner section* can be called to fetch the first input symbol.

5.2.2 Function: *pm*ISym()

```
pmISym(code, symString, printString, token)

code: IN %%(codeType)
symString: IN STRING
printString: IN STRING
token: OUT %%(tokenType)
```

Called from the error recovery routines when the parser has decided upon inserting a symbol in the input stream in order to recover from a syntax error. It is intended for error reporting, and for constructing a semantically feasible token. The parser error recovery process has no notion at all about semantics, it just decides upon a syntactically reasonable token to insert. Thus the parser implementor must supply code to construct a token that has semantic meaning to the application.

Consider the following example: when the scanner reads an `INTEGER`, a normal action for the scanner is to compute the value of the integer read and return it as a terminal attribute. When the parser error recovery process decides to insert an `INTEGER` token in the input string, it has no knowledge about how the scanner handles an `INTEGER`, so in order to construct a semantically feasible token we must set the value attribute to some value for an `INTEGER` in *pm*`ISym()`.

The parameters: `code` is the terminal code for the inserted token, `symString` is the terminal string as used in the grammar, `printString` is the terminal print string as defined in the grammar, and `token` is the token that should be created.

This function is defined in the generated error recovery interface module, *pm*Err, if selected by use of the `errorhandler` option. The actual code to construct a semantically valid token should be defined in the ParserMaker description file (see *The Insertsymbol Section* on page 97). The code from the *insertsymbol section* is inserted into the body of the *pm*`ISym()` function if the option `Errorhandler;` is set, otherwise this functionality must be supplied by the implementor.

5.2.3 Function: *pm*DSym()

```
pmDSym(token, symString, printString)

token: IN %%(tokenType)
symString: IN STRING
printString: IN STRING
```

Called from the error recovery routines when the parser has decided upon deleting a symbol from the input stream in order to recover from a syntax error. It is intended for error reporting and clean up. The first parameter is the deleted `token`. The `symString` and `printString` has the same interpretation as for *pm*`ISym()`. The function is defined in the default error recovery

interface module that is automatically generated if the `Errorhandler` option is set.

The actual code to necessary to clean up may be defined in the ParserMaker description file (see *The Deletesymbol Section* on page 97). The code may for example return dynamically allocated areas referenced by the semantic attributes of the token. The code from the *deletesymbol section* is inserted into the body of the ***pm***`DSym()` function if the option `Errorhandler;` is set, otherwise this functionality must be supplied by the implementor.

5.2.4 Function: ***pm***RPoi()

```
pmRPoi(token)

token: IN %%(tokenType)
```

Called from the error recovery routines when an error has been recovered from. It is intended for error reporting. The parameter is the `token` from which the parsing process will continue. The default module ***pm*Err** implementing the function is generated if the `Errorhandler` option is set, otherwise the function must be supplied by the implementor.

5.2.5 Function: ***pm***Mess()

```
pmMess(token, method, code, severity)

token: IN %%(tokenType)
method: IN INTEGER
code: IN INTEGER
severity: IN INTEGER
```

Called from the error recovery routines when the syntax error has been repaired, and an error message should be output. The function is generated in the error recovery interface module if the `Errorhandler` option is set. Otherwise the function must be supplied by the implementor.

The parameters: `token` is the token which activated the error recovery process, `method` is the recovery method applied, `code` is an error classification, and `severity` is the error severity code.

The values of the integer arguments have the following meaning:

Method:
- 1 = Symbol(s) insertion
- 2 = Symbol(s) deletion
- 3 = Symbol(s) replacement
- 4 = Stack backup
- 5 = Halted

Code:
- 1 = Unknown token received from scanner
- 2 = Syntax error

3 = Parse stack overflow
4 = Parse table error

Severity:

1 = Warning
2 = Error (recoverable)
3 = Fatal error
4 = System error & Limit error

A THE PL/0 EXAMPLE

This example is a part of the pl/0 example used throughout the ToolMaker documentation. The *ToolMaker System Description* contains a detailed walk-through of the relevant parts of the example. This appendix contains the files relevant for ParserMaker.

A.1 pl0.pmk - The ParserMaker Description File

```
------------------------------------------------------------
-- pl0.pmk    Date: 1993-12-27/toolmake
--
-- pl0 - ParserMaker grammar description file
--
------------------------------------------------------------
-- Created:   1993-04-27/reibert@roo
-- Generated: 1993-12-27 15:39:38/toolmake v2,r0,c4
------------------------------------------------------------

%%OPTIONS
     Verbose;
%%END

%%IMPORT
#include <stdio.h>
#include <strings.h>

#include "pl0Parse.h"
#include "pl0List.h"
#include "pl0Scan.h"
extern pl0ScContext lexContext;

%%EXPORT
/* Semantic attribute interface */

/* Internal node type */
typedef struct Node {
    char *string;
    int value;
    struct Node *next;
} Node;

%%DECLARATIONS


/* Counters for the various contructs */
static int constants, variables, procedures, statements;

/* The node constituting the program */
static Node *program;


/*------------------------------------------------------------
    makeNode()
    Make a new list node and initialize it.

 */

static Node *makeNode(char *string, int value)
{
```

```
    Node *temp;

    temp = (Node *)malloc(sizeof(Node));
    temp->string = (char *)malloc(strlen(string) + 1);
    strcpy(temp->string, string);
    temp->value = value;
    temp->next = NULL;
    return temp;
}


/*-------------------------------------------------------
    append()

    Append two lists of nodes.

 */
static Node *append(Node *list1, Node *list2)
{
   if (list1->next) return append(list1->next, list2);
    else {
     list1->next = list2;
     return list1;
    }
}


/*-------------------------------------------------------
    summary()

    Print a summary of the PL/0 program using the Lister.

 */
void summary()
{
  Node *node;
  char buf[256];

  sprintf(buf, "Number of constants:  %u", constants);
  pl0LiPrint(buf);
  for (node = program; node; node = node->next) {
    sprintf(buf,"  %s = %d", node->string, node->value);
    pl0LiPrint(buf);
  }
  pl0LiPrint("");
  sprintf(buf, "Number of variables:  %u", variables);
  pl0LiPrint(buf);
  sprintf(buf, "Number of procedures: %u", procedures);
  pl0LiPrint(buf);
  sprintf(buf, "Number of statements: %u", statements);
  pl0LiPrint(buf);
}


%%SCANNER

     scan(token);

%%TERMINALS
     IDENTIFIER= 2,1,1 => '<id>';
     NUMBER    = 3, 1,1 => '0';
     '.'       = 4;
     ';'       = 5;
```

```
    ','       = 6;
    '='       = 7;
    '<'       = 9;
    '>'       = 9;
    '<>'      = 10;
    '<='      = 11;
    '>='      = 12;
    '('       = 13;
    ')'       = 14;
    '+'       = 15;
    '-'       = 16;
    '*'       = 17;
    '/'       = 18;
    ':='      = 33;
    'IF'      = 34;
    'DO'      = 35;
    'CALL'    = 36;
    'VAR'     = 25;
    'END'     = 26;
    'ODD'     = 27;
    'THEN'    = 28;
    'CONST'   = 29;
    'BEGIN'   = 30;
    'WHILE'   = 31;
    'PROCEDURE'= 32;
    INCLUDE   = 37;     -- Include is allowed but
                        -- handled by the scanner


%%ATTRIBUTES
    srcp %% TmSrcp %1 %%,
   -- Extra user-defined fields and attributes should be
   -- added here
    list %% Node* %1 %%,
    stringValue %% char %1[256] %%,
    integerValue %% int %1 %%;

%%RECOVERY
   Fiducial('CONST', 'VAR', 'PROCEDURE', 'CALL', 'BEGIN',
            'IF', 'WHILE');
   Separator(',', ';');

%%RULES

<program>
     =
       <block> '.'
          %%
            program = %<block>.list;
          %%
     ;

<block>
     = <declarations> <statement>
       -- propagate the constant list
       %%
         %<block>.list = %<declarations>.list;
       %%
     ;

<declarations>
     = <constant declaration> <variable declaration>
             <procedure declarations>
```

```
        -- propagate the constant list
        %%
          %<declarations>.list =
                        %<constant declaration>.list;
        %%
      ;
<constant declaration>
      = 'CONST' <constant definitions> ';'
        %%
          %<constant declaration>.list =
                        %<constant definitions>.list;
        %%
      ! -- Empty
        %%
          %<constant declaration>.list = NULL;
        %%
      ;

<constant definitions>
      = <constant definitions> ',' <constant definition>
        %%
          %1<constant definitions>.list =
           append(%2<constant definitions>.list,
               makeNode(%<constant definition>.stringValue,
                     %<constant definition>.integerValue));
        %%
      ! <constant definition>
        %%
          %<constant definitions>.list =
           makeNode(%<constant definition>.stringValue,
                 %<constant definition>.integerValue);
        %%
      ;

<constant definition>
      = IDENTIFIER '=' NUMBER
        %%
          constants++;
          strcpy(%<constant definition>.stringValue,
                 %IDENTIFIER.stringValue);
          %<constant definition>.integerValue =
                     %NUMBER.integerValue;
        %%
      ;

<variable declaration>
      = 'VAR' <identifiers> ';'
      ! -- Empty
      ;

<identifiers>
      = <identifiers> ',' IDENTIFIER
        %%
          variables++;
        %%
      ! IDENTIFIER
        %%
          variables++;
        %%
      ;

<procedure declarations>
```

```
        = <procedure declarations> <procedure declaration>
        ! -- Empty
        ;

<procedure declaration>
        = 'PROCEDURE' IDENTIFIER ';' <block> ';'
          %%
            procedures++;
          %%
        ;

<statement>
        = <assignment statement>
          %%
            statements++;
          %%
        ! <call statement>
          %%
            statements++;
          %%
        ! <compound statement>
          %%
            statements++;
          %%
        ! <if statement>
          %%
            statements++;
          %%
        ! <while statement>
          %%
            statements++;
          %%
        ! -- Empty
        ;

<assignment statement>
        = IDENTIFIER ':=' <expression>
        ;

<call statement>
        = 'CALL' IDENTIFIER
        ;

<compound statement>
        = 'BEGIN' <statements> 'END'
        ;

<statements>
        = <statements> ';' <statement>
        ! <statement>
        ;

<if statement>
        = 'IF' <condition> 'THEN' <statement>
        ;

<while statement>
        = 'WHILE' <condition> 'DO' <statement> ;


<condition>
        = 'ODD' <expression>
        ! <expression> <relop> <expression>
```

```
     ;

<relop> = '=' ! '<>' ! '<' ! '>' ! '<=' ! '>=' ;

<expression>
     = <optional sign> <terms>
     ;

<terms>
     = <terms> <addop> <term>
     ! <term>
     ;

<term>
     = <term> <mulop> <factor>
     ! <factor>
     ;

<factor>
     = IDENTIFIER
     ! NUMBER
     ! '(' <expression> ')'
     ;

<optional sign> = '+' ! '-' ! ;        --Note the empty
                                       --last alternative!

<addop> = '+' ! '-' ;

<mulop> = '*' ! '/' ;
```

B ERROR MESSAGES

Error messages produced by ParserMaker are output to the terminal, and if the option `List Input;` is set, also to the list file.

For a description of the format of error messages refer to *Message Format* in the *ToolMaker System Description*, page 49.

B.1 Message Explanations

The following list gives a brief summary of the error messages and in some cases the actions that should be taken by the user. Messages with numbers less than 100 are messages common for all Makers. These are described in *Messages Explanations* in the *ToolMaker System Description*, page 50. Messages indicating license problems or limitations are described in *License Errors* in the *ToolMaker System Description*, page 51.

`100 I Parsing resumed.`

Issued after a syntax error has resulted in some symbols being deleted. The `sequence number` indicates the point from which scanning and parsing will continue.

`101 E %1 inserted.`

Syntax error, the recovery method has inserted some symbol(s) in order to be able to continue parsing. The syntax error(s) must be corrected in order for a valid result to be produced.

`102 E,F %1 deleted.`

Syntax error, the recovery method has deleted some symbol(s) in order to be able to continue parsing. The syntax error(s) must be corrected in order for a valid result to be produced.

`103 E %1 replaced by %2.`

Syntax error, the recovery method has replaced some symbol(s) in order to be able to continue parsing. The syntax error(s) must be corrected in order for a valid result to be produced.

`104 E Syntax error, stack backed up.`

Severe syntax error, the recovery method has been forced to pop symbols off the stack in order to be able to continue. The syntax error(s) must be corrected in order for a valid result to be produced.

`106 S Parse stack overflow.`

The ParserMaker internal parse stack has overflowed. Please save the files that have caused the error and contact your ToolMaker contact person.

```
107 S Parse table error.
```

Errors have been detected in the ParserMaker internal parse tables. Please save the files that have caused the error and contact your ToolMaker contact person.

```
108 I Parsing terminated.
```

Issued when it is impossible for ParserMaker to make any sense of the input. The error messages preceding this message will hopefully give some hint to what is wrong.

```
113 E Reduction and no-reduction sets intersect. Set = {%1}.
```

In the indicated production there is both a *reduce-for* and a *reduce-not-for* clause for the indicated symbols. Decide what action you want on the indicated symbols and delete them from the other clause.

```
114 W Repetition of Null string ignored.
```

The EBNF-construct for repetition has been used, but it contains no symbols and is thus not necessary. If the offending rule is correct without the repetition, remove the repetition, otherwise correct the rule.

```
115 E Ambiguous symbol %1 ignored, terminal or non-terminal
      expected.
```

A terminal or a non-terminal are the only symbols allowed in this position, and the found symbol was neither.

```
116 E Ambiguous symbol %1, non-terminal expected.
```

A non-terminal is the only symbol allowed in this position, and the found symbol was not a non-terminal.

```
117 E Ambiguous symbol %1, terminal expected.
```

A terminal is the only symbol allowed in this position, and the found symbol was not a terminal.

```
118 E Ambiguous symbol %1, attribute expected.
```

An attribute is the only symbol allowed in this position, and the found symbol was not an attribute.

```
119 W Attribute %1 defined twice.
```

The indicated attribute has been multiply defined in the *attributes section*. Remove all but one definition.

```
120 E Terminal code %1 already in use.
```

The indicated terminal code has been used for more than one terminal. Each terminal must have a unique terminal code. Note that terminal codes 0 and 1 are reserved by ParserMaker and must not be used.

121 W Terminal %1 defined twice, old definition overridden.

The indicated terminal has been defined more than once in the *terminals section*. The last definition is the one used by the system. Remove all but one definition.

123 S Ambiguous symbol %1, terminal expected.

A symbol has been found in the *terminals section* that ParserMaker thinks is not a terminal. Please save the files that have caused the error and contact your ToolMaker contact person.

130 F End of file in %1.

End of file has been found in a target code section. Truncated file or missing '%%'?

131 F Quoted string in semantic action does not terminate.

A single-quoted string following a '%'-sign (i.e. a ParserMaker symbol) does not terminate. You have probably forgotten the ending quote, and thus the rest of the semantic action has been skipped.

148 E Repetition construct contains only modifiers or semantic actions.

The EBNF repetition construct has been used in a production, but it contains no grammar symbols and is thus meaningless. Remove the repetition construct or correct the rule.

149 E Symbol not recognized in production.

The indicated grammar symbol used in the attribute reference does not occur in the production and is thus erroneous. Misspelling or an attempt to use attribute references in semantic actions embedded in a production (this is not allowed, see *Grammar Attributes* on page 64 for further details).

150 E Undefined attribute %1.

The indicated attribute has not been defined in the *attributes section*. Misspelling?

151 E Reference to outer EBNF construct is not possible.

An attribute to an outer EBNF (OEBNF) construct has been made, but no outer EBNF construct exists in this context, so a reference is not possible.

152 E Attribute %1 must be non-terminal or vocabulary attribute.

A reference to a terminal attribute has been made, but for this symbol only non-terminal or predefined attributes are allowed.

153 E Wrong instance in attribute reference (OEBNF), first instance assumed.

The indicated instance of the outer EBNF symbol does not exist in this context.

155 E %1 expected.

The indicated symbol is not allowed in this context, instead the symbol given in the error message is expected.

156 E EBNF expression instance not recognized in production.

The indicated instance of the outer EBNF (OEBNF) symbol does not exist in this context.

158 E Attribute class in conflict with symbol type.

A terminal attribute has been referenced for a non-terminal or vice versa.

159 E Symbol instance not recognized in production.

The indicated instance of the grammar symbol does not exist in this context.

166 E You are not allowed to use both the %1 and the PACK directives.

The packing of tables are selected either using the Pack directive which influences the packing of both the action and goto tables simultaneously, or using the Actionpack and Gotopack directives which influences the packing separately. If you have specified Actionpack you *must* use Gotopack if you want to specify the packing of the goto tables.

196 F Cannot allocate memory for hash table.

197 F Cannot allocate memory for entry in vocabulary table.

201 F Cannot allocate memory for production data structure.

204 F Cannot allocate memory for nodes to build syntax tree.

205 F Cannot allocate memory for modification data structure.

206 F Cannot allocate memory for attribute reference data structure.

207 F Cannot allocate memory for semantic actions table.

208 F Cannot allocate memory for attribute storage structure.

209 F Cannot allocate memory for strings to generate non terminal.

213 F Cannot allocate memory for sorting grammar.

214 F Cannot allocate memory for strings to make error message.

Failure to allocate memory for the indicated data structure. Contact your system administrator for advice on how to obtain more memory.

```
215 S Production queue is empty.
```

The ParserMaker internal production processing queue has been emptied abnormally. Please save the files that have caused the error and contact your ToolMaker contact person.

```
230 W %1 given in the TERMINALS SECTION but never used in
    grammar.
```

A terminal has been explicitly given in the *terminals section*, but it does not appear in any production, so it is ignored. Misspelling?

```
231 I %1 was not included in the TERMINALS SECTION.
```

The indicated terminal has been used in the grammar, without being defined in the *terminals section*. Since this is perfectly legal, and sometimes desirable, one way to get rid of this message is to specify the directive `List` without the `info` option.

```
232 W More than one goal symbol found, %1 used.
```

More than one non-terminal fulfils the conditions for being chosen as goal symbol, so the grammar is probably erroneous. Misspelling? Remember that symbols are case sensitive!

```
233 W No explicit goal symbol found, %1 used.
```

No non-terminal fulfils the condition that it should never appear in any right hand side, so the indicated non-terminal (the left hand side of the first production) is chosen by ParserMaker as the goal symbol. If your goal symbol appears on the right hand side of at least one production, make sure that a rule deriving your goal symbol is the first production of the grammar!

```
234 E Grammar is non terminating, symbol %1.
```

The grammar has infinite recursion, e.g. the definition of symbol `X` includes `X` in all its right hand sides:

```
X = X Y Z
    ! T X V
    ;
```

Note that the symbol printed in the error message is not necessarily the offending symbol, instead it is the top symbol in the infinite derivation tree. Thus all symbols that can be derived from the printed symbol in one or more steps should be checked!

```
235 E Section already defined.
```

You may only specify the various sections once in the description file.

```
236 E No sections allowed after %%RULES-section.
```

The *rules section* must be the last section in the description file.

```
237 W %%INSERTSYMBOL and/or %%DELETESYMBOL sections used
    although errorhandler generation turned off.
```

One or both of the *insertsymbol* and *deletesymbol sections* where present in the description file. The option 'No Errorhandler;' was also used to turn the generation of an errorhandler off. This means that the target code in the sections will not be used.

```
301 W Conflicts resolved by default rules.
```

Default rules have been used to resolve LALR(1) conflicts. The list file contains details about the changes that have been made.

```
302 W Conflicts resolved by modifications.
```

Modifications have been used to resolve ambiguities. The log file contains information about the effects.

```
303 E Grammar is not LALR(1).
```

The grammar is ambiguous, and the default disambiguating rules and/or the supplied modifications were not sufficient to disambiguate the grammar. Inspect the log file (see *Format of the Generated Item Set* on page 100) and modify the grammar. If the grammar is hard to make LALR(1), try using more disambiguating rules.

```
401 F Description file "%1" not found.
```

The indicated description file could not be found. Some possible causes:

- You misspelled the file name.
- You are in the wrong directory.
- You did not take into account that **.pmk** is added to files without any extension.

```
402 F Could not open list file "%1".
403 F Could not open table file "%1".
404 F Could not open vocabulary file "%1".
```

The indicated output file could not be opened. The most probable cause is that you do not have write permission on the directory or that the file already existed and is write protected.

C DESCRIPTION LANGUAGE

This appendix contains a syntax summary of the ParserMaker description language. For a full discussion of the language, see *The ParserMaker Description File* on page 79.

For a brief discussion of the notation used in this description see appendix C in the *ToolMaker System Description, SYNTAX NOTATION*, page 52.

C.1 Language Syntax

```
<description file> ::=
     [ <toolmaker sections> ]
     [ <other sections> ]
     <rules section>

<toolmaker sections> ::=
     [ <options section> ]
     { <import section> | <srcp section>
          | <token section> }

<other sections> ::=
     { <declarations section>
     | <terminals section>
     | <attributes section>
     | <recovery section>
     | <export section>
     | <scanner section>
     | <insertsymbol section>
     | <deletesymbol section> }

<options section> ::=
     ['%%OPTIONS' <directive> {<directive>} ['%%END']]

<directive> ::=
       <common directive>
     | <listerprefix directive>
     | <errorhandler directive>
     | <trace directive>
     | <lookaheadmax directive>
     | <shiftcost directive>
     | <stacklimit directive>
     | <pack directive>
     | <list directive>
     | <optimize directive>
     | <recovery directive>
     | <resolve directive>
     | <vocabulary directive>
     | <table file directive>
     | <list file directive>

<common directive> ::=
       <target directive>
     | <os directive>
     | <prefix directive>
     | <library directive>
     | <escape directive>
     | <width directive>
     | <height directive>
```

```
      | <generate directive>
      | <force directive>

<listerprefix directive> ::=
     'LISTERPREFIX' <quoted string> ';'

<errorhandler directive> ::=
     ['NO'] 'ERRORHANFLER' ';'

<trace directive> ::=
     ['NO'] 'TRACE' ';'

<lookaheadmax directive> ::=
     'LOOKAHEADMAX' <number> ';'

<shiftcost directive> ::=
     'SHIFTCOST' <number> ';'

<stacksize directive> ::=
     'STACKLIMIT' <number> ';'

<pack directive> ::=
       'PACK' <pack option>
              {',' <pack option>} ';'
      | ['NO'] 'PACK' ';'
      | 'ACTIONPACK' <pack option>
              {',' <pack option>} ';'
      | 'GOTOPACK' <pack option>
              {',' <pack option>} ';'

<pack option> ::=
     'ROW' | 'COLUMN' | 'RDS' |
     'GCS' | 'LES' | 'ERROR'
<list directive> ::=
       'LIST' <list option>
              {',' <list option>} ';'
     | ['NO'] 'LIST' ';'

<list option> ::=
       'INPUT' | 'GRAMMAR' | 'ITEMS'
     | 'TABLES' | 'STATISTICS' | 'INFO'

<optimize directive> ::=
       'OPTIMIZE' [<optimize option>
                {',' <optimize option>}] ';'
     | ['NO'] 'OPTIMIZE' ';'

<optimize option> ::= 'LR0'

<recovery directive> ::=
       'RECOVERY' [<error option>
                {',' <error option>}] ';'
     | ['NO'] 'RECOVERY' ';'

<error option> ::=
       'SINGLE' | 'MULTIPLE' | 'PANIC'

<resolve directive> ::=
       'RESOLVE' <resolve option>
                {',' <resolve option>} ';'
     | ['NO'] 'RESOLVE' ';'
```

```
<resolve option> ::=
       'SR' | 'RR'

<import section> ::=
     '%%IMPORT' <target language code> ['%%END']

<export section> ::=
     '%%EXPORT' <target language code> ['%%END']

<declarations section> ::=
     '%%DECLARATIONS' <target language code> ['%%END']

<terminals section> ::=
     '%%TERMINALS' {<terminal definition> ';'}
     ['%%END']

<terminal definition> ::=
     TERMINAL '=' <token code> [<error recovery data>]

<token code> ::= <number>

<error recovery data> ::=
     ',' <insert cost> ',' <delete cost> [<print symbol>]

<insert cost> ::= <number>

<delete cost> ::= <number>

<print symbol> ::= '=>' TERMINAL

<attributes section> ::=
     '%%ATTRIBUTES'
     <grammar attribute> {',' <grammar attribute>} ';'
     ['%%END']

<grammar attribute> ::=
     <identifier> [ '%%' <target language code> '%%' ]

<scanner section> ::=
     '%%SCANNER' <target language code> ['%%END']

<insertsymbol section> ::=
     '%%INSERTSYMBOL' <target language code> ['%%END']

<deletesymbol section> ::=
     '%%DELETESYMBOL' <target language code> ['%%END']

<recovery section> ::=
     '%%RECOVERY'
     [<meta part>]
     [<separator part>]
     [<fiducial part>]
     [<skip part>]
     ['%%END']

<meta part> ::=
     'META' {<meta name> '=' '(' TERMINAL
          {',' TERMINAL} ')' ['=>' TERMINAL] ';'}

<separator part> ::=
     'SEPARATOR' '(' TERMINAL {',' TERMINAL} ')' ';'
```

```
<fiducial part> ::=
     'FIDUCIAL' '(' TERMINAL {',' TERMINAL} ')' ';'

<skip part> ::=
     'SKIP' '(' TERMINAL {',' TERMINAL} ')' ';'

<rules section> ::=
     '%%RULES'
     {NONTERMINAL '=' <alternatives> ';'}
     ['%%END']

<alternatives> ::=
     <right hand side> {'!' <right hand side>}

<right hand side> ::= <components> <opt modify>

<components> ::= {<component>}

<component> ::=
       <symbol>
     | <action modify>
     | '(' <components> {'|' <components>} ')'
     | '{' <components> '}'
     | '[' <components> ']'

<symbol> ::= TERMINAL | NONTERMINAL

<action modify> ::=
     <opt modify> <semantic action> <opt modify>

<semantic action> ::=
     '%%' <any character sequence, but %%> '%%'

<opt modify> ::=
     {('%+' | '%-') '(' TERMINAL {',' TERMINAL} ')'}
```

C.2 Lexical Items

```
TERMINAL ::=
       <identifier>
     | <angle bracketed string>
     | <quoted string>

NONTERMINAL ::=
       <identifier>
     | <angle bracketed string>

ATTRIBUTE ::=
       <identifier>

<identifier> ::=
       <letter> {<letter> | <digit> | '_'}

<string> ::=
       <quoted string>
     | <angle bracketed string>

<quoted string> ::=
     ''' <character> {<character>} '''

<angle bracketed string> ::=
     '<' <character> {<character>} '>'
```

```
<number> ::= <decimal integer> | <hexadecimal integer>

<decimal integer> ::= <digit> {<digit>}

<hexadecimal integer> ::= '#' <hex digit> {<hex digit>}

<hex digit> ::= <digit>
              | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
              | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'

<letter> ::=
     <upper case letter> | <lower case letter>

<special character> ::=
       '!' | '#' | '$' | '%' | ''' | '(' | ')' | '+'
     | ',' | '-' | '.' | ';' | '<' | '=' | '>' | '_'
     | '&' | '{' | '}' | '|' | '\' | '[' | ']'
```

D TARGET LANGUAGE DETAILS

In the descriptions below, the IMP insert statements `%%(tokenType)` and `%%(codeType)` refers to the type names defined by the token section (see *The Token Section* in the *ToolMaker System Description*, page 36).

D.1 'c'

```
void pmParse();

void pmRPoi(token)
     %%(tokenType) *token;

void pmISym(code, symString, printString, token)
     %%(codeType) code;
     char *symString;
     char *printString;
     %%(tokenType) *token;

void pmDSym(token, symString)
     %%(tokenType) *token;
     char *symString;

void pmMess(token, method, code, severity)
     %%(tokenType) *token;
     int method;
     int code;
     int severity;
```

D.2 'ansi-c' and 'c++'

```
void pmParse();

void pmRPoi(%%(tokenType) *token);

void pmISym(%%(codeType) code,
            char *symString,
            char *printString,
            %%(tokenType) *token);

void pmDSym(%%(tokenType) *token,
            char *symString);

void pmMess(%%(tokenType) *token,
            int method,
            int code,
            int severity);
```

Part III

ScannerMaker Reference Manual

1 INTRODUCTION 133

2 CONCEPTS AND ASSUMPTIONS 134

2.1 ScannerMaker and Scanner 134
2.2 Vocabulary 134
2.3 Token 134
2.4 Scanner 134
2.5 Scanner and Inheritance 134
2.6 Screening 134
2.7 Token and Source Position Definitions 135

3 SCANNER PRODUCTION 136

3.1 Creating a Running Scanner 136
3.1.1 File Descriptions 138
3.2 The ScannerMaker Description File 138
3.2.1 Lexical items 138
3.2.2 Overall Structure 139
3.2.3 The Options Section 140
3.2.4 The Declaration Section 145
3.2.5 The Context Section 145
3.2.6 The Export Section 145
3.2.7 The Reader Section 146
3.2.8 The Action Section 146
3.2.9 The Prehook Section 147
3.2.10 The Posthook Section 147
3.2.11 The Set Section 148
3.2.12 The Map Section 149
3.2.13 The Definition Section 150
3.2.14 The Vocabulary Section 151
3.2.15 The Scanner Subsection 151
3.2.16 The Screened Token Subsection 152
3.2.17 The Undefine Token Subsection 152
3.2.18 The Rules Subsection 152
3.2.19 The Skip Subsection 153
3.2.20 Regular Expressions 154
3.2.21 Semantic Actions 158
3.3 The ToolMaker Common Description file 160
3.4 The Vocabulary file 160

4 THE SCANNERMAKER COMMAND 162

4.1 Parameters 162
4.2 Options 162

1 INTRODUCTION

ScannerMaker is a tool which provides a simple and yet powerful way of creating fast and robust scanners.

The main features of ScannerMaker are:

- Multiple vocabularies.
- Multiple scanners.
- Dynamic token size.
- Creates very fast scanners.
- User defined semantic actions associated with tokens.
- Generate scanners for different character sets.
- Generated scanners are able to handle nested levels of input, e.g. include files in programming languages.
- Integrated with other parts of ToolMaker.
- Various packing schemes to optimize for speed or space.

The reader must have a fairly well grounded knowledge in the area of scanners to take full advantage of this document. The concept of regular expressions, semantic actions, and tokens should be well understood before reading this manual.

2.7 Token and Source Position Definitions

The generated scanner supports automatic source position calculation. The *row*, *column* and *position* source position members are automatically updated by the generated scanner if they are defined in the *source position section* is the ToolMaker Common Description file (**.tmk**). See *The Token Section* on page 36 and *The Srcp Section* in the *ToolMaker System Description*, page 34.

Depending on the target language used the produced target language module, ***sm*Scan**, may be one or more files, e.g. for 'c' there will be one ***sm*Scan.c** and one ***sm*Scan.h** created.

The only files that must be maintained by the user are ***sm*.smk** and ***sm*.tmk**. A vocabulary file is not necessary though it is strongly recommended that a vocabulary file is used.

The following walk-through should give you a feeling how to create an operational scanner, step by step.

1) Create a ToolMaker description file (***sm*.tmk**) according to the specifications in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. This step is not necessary if you are developing a stand-alone scanner (in this case the information can be put inside ***sm*.smk** instead).

2) Create a ScannerMaker description file (***sm*.smk**) according to the specifications in *The ScannerMaker Description File* on page 138.

3) Either run ***make*** and skip to step 6, or run ***smk***, ScannerMaker, with ***sm*.smk** as argument. ScannerMaker can be controlled by using various options as described in *THE SCANNERMAKER COMMAND* on page 162.

4) Compile the ScannerMaker generated source module, ***sm*Scan**, the scanner semantic actions module ***sm*ScSema**, and the common definition module ***sm*Common**. If the target language was 'c', ***sm*Scan.c**, ***sm*Scan.h**, ***sm*ScSema.c** and ***sm*Common.h** were produced.

5) Compile and link the main program and any auxiliary modules with the object files produced.

6) Execute and test the scanner.

Points 1 and 2 can be simplified by invoking ***toolmake*** to set up a development environment for you (see the *Toolmake Reference Manual*). The files created by ***toolmake*** must be modified to suit your requirements.

> *Note:* In rare cases the description files are not sufficient to capture all the target dependencies or special requirements for a project, and you have to modify the scanner skeletons. If this is the case copy the file **Scan.imp** to the development directory from the appropriate library and modify them. TO DO THIS IT IS ABSOLUTELY NECESSARY THAT YOU HAVE A THOROUGH UNDERSTANDING OF THE SYSTEM. Change the library option in the ScannerMaker description file to point to the directory where the modified skeletons resides. When ScannerMaker is executed, it will use these skeletons instead of the standard ones.

```
    | '}' | '[' | ']' | ';' | ':' | ',' | '"' | '~'
    | '`' | '<' | '>' | ',' | '.' | '?' | '/' | '\'

<target code> ::=
     <any characters in the target language except '%%'>

<token name> ::=
     <letter> {<letter> | <digit> | '_'}

<definition name> ::=
     <letter> {<letter> | <digit> | '_'}

<string> ::=
     ''' {<letter> | <digit> | <special character>} '''
```

3.2.2 Overall Structure

The overall structure of the Scanner Description file is as follows:

```
<description file> ::=
     <toolmaker sections>
     {<target code section>}
     {<set definition section>}
     {<general definition section>}
     {<vocabulary section>}
```

The *toolmaker sections* are further described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30.

```
<toolmaker sections> ::=
     [ <options section> ]
     { <import section> | <srcp section>
          | <token section> }
```

The option section follows the general guidelines as described in *The Options Section* in the *ToolMaker System Description*, page 30. The options which can be specified in the options section are described in *Options* on page 162. The *import*, *srcp* and *token sections* are described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30.

The *target code sections* are one of the following subsections:

```
<target code section> ::=
       <declaration section>
     | <context section>
     | <export section>
     | <reader section>
     | <prehook section>
     | <posthook section>
     | <action section>
```

The *general definition sections* are one of the following subsections:

```
<general definition section> ::=
       <map definition section>
     | <definition section>
```

The *vocabulary sections* have the following structure:

MAKER COMMAND on page 162). If an option is not specified its default value is used. The default options are:

```
No Verbose;
Target 'ansi-c';
Os 'SunOS';
Prefix 'sm';
Library '$TMHOME/lib/ansi-c';
Escape '`';
Width 78;
Height 60;
Generate source;
No Force;
Optimize;
No Trace;
Set 'ISO8859_1';
Pack row, column;
No Screening;
No List;
Tokensize 1024;
Tokenlimit 524288;
```

Note however that the settings of common options in the ToolMaker Common Description file influences the default values (see *The Options Section* in the *ToolMaker System Description*, page 30).

3.2.3.1 Common Directives

```
<common directive> ::=
        <target directive>
      | <os directive>
      | <prefix directive>
      | <library directive>
      | <escape directive>
      | <width directive>
      | <height directive>
      | <generate directive>
      | <force directive>
```

The common directives are directives available for all of the Makers in the ToolMaker kit. For a detailed description of these refer to *The Options Section* in the *ToolMaker System Description*, page 30. All directives are available for ScannerMaker, and if used overrides settings and default values from the ToolMaker Common Description file.

The prefix directive does not inherit its default value, instead it defaults to '***sm***' if not explicitly set in the **.tmk** file. If set in the ToolMaker Common Description file and *not* used in the ScannerMaker Description file it defaults to the system prefix (the value set in the ToolMaker Common Description file).

3.2.3.2 The Optimize Directive

```
<optimize directive> ::=
     ['NO'] 'OPTIMIZE' ';'
```

```
Pack row, column;
```

Each packing scheme may be combined freely with any other packing scheme with the only exception of ERROR which only has significance together with GCS or LES packing schemes. For example:

```
PACK LES,RDS,COLUMN;
```

instructs ScannerMaker to first locate equivalent columns in the state table and merge them. Thereafter a line elimination scheme is used to further reduce the size of the packed table and finally row displacement is used to minimize the table still further.

The order in which the packing is performed is

1) ROW and/or COLUMN

2) LES

3) GCS

4) RDS

Which packing scheme to use depends on if speed or space requirements is of greatest importance. The same packing scheme may pack some state tables better than other tables. Generally, the best packing schemes are ROW and COLUMN, or RDS, while LES and GCS often gives a good result but are rather slow mainly because of the need of an error state table.

If LES is used it is recommended that GCS is also used because the use of GCS does not affect the speed. The use of RDS is also recommended when LES and/or GCS is used to further enhance packing without greatly affecting the execution speed. However, if speed is essential no packing at all, ROW or COLUMN packing or RDS packing should be used.

3.2.3.6 The Screening directive

```
<screening directive> ::=
        'NO' 'SCREENING' ';'
      | 'SCREENING' <minimum token size> ';'
```

Screening is used to reduce the size of the generated scanners. Screening removes the need for all special states for rules which is defined as a sequence of characters, a stream of characters, with a specified minimum length and for which there is another rule which accepts the same token. The rule which is removed defines a *screened* token and the more general rule which accepts the same token is said to *screen* a screened token.

When a token which screens another token is found a look-up is performed to check if there is a screened token which is equal to the token found. For example:

3.2.3.9 The Token Limit Directive

```
<token directive> ::=
     'TOKENLIMIT' <maximal token size> ';'
```

The token scanned may not be larger than the specified maximal token length. By the default this is set to

```
Tokenlimit 524288;
```

3.2.3.10 The Exclude Character Directive

```
<exclude character directive> ::=
     'EXCLUDE' <exclude character> ';'
```

This option specifies a character excluded from the normal character set. The specified character may never occur in the input and can be used for special purposes by ScannerMaker to create an even more efficient scanner. The default is

```
No Exclude;
```

I.e not to exclude any character from the selected character set.

3.2.4 The Declaration Section

```
<declaration section> ::=
     '%%DECLARATION' <target language code> ['%%END']
```

In this section types, variables and functions used within the scanner should be defined. This target language dependent source code is copied into the generated scanner and is accessible (valid) in all other target language sections.

3.2.5 The Context Section

```
<context section> ::=
     '%%CONTEXT' <target language code> ['%%END']
```

In this section variables used within a scanner context should be defined, see *Semantic Actions* on page 158 and *Type: smScContext and smScContextItem* on page 164 for a complete description of how to use the scanner context.

3.2.6 The Export Section

```
<export section> ::=
     '%%EXPORT' <target language code> ['%%END']
```

User defined functions and variables that should be visible outside the generated scanner should be defined in this section, this section is included in the interface description of the generated scanner (in 'c' the ***sm*Scan.h** file).

3.2.9 The Prehook Section

```
<prehook section> ::=
     '%%PREHOOK' <target language code> ['%%END']
```

In the *prehook section* code which should be performed before scanning a token is defined. For more information on the contents of this section, see *Semantic Actions* on page 158. The *prehook section* may be viewed as the body of a function with the following definition

```
code = smScPreHook(smThis,smToken)

smThis    : IN smScContext
smToken   : IN OUT %%(tokenType)
returns INTEGER
```

If a positive number, zero included, is returned the scanning is terminated immediately. The number is used as the external token code returned by the scanner.

> *Note:* When executing the prehook the variable `smLength` has the value 0 (zero).

3.2.10 The Posthook Section

```
<posthook section> ::=
     '%%POSTHOOK' <target language code> ['%%END']
```

The code specified in the *posthook section* is executed after a complete token is found, see *Semantic Actions* on page 158 for more information on the contents of this section. The *posthook section* may be viewed as the body of a function with the following definition:

```
code = smScPostHook(smThis,smToken)

smThis    : IN smScContext
smToken   : IN OUT %%(tokenType)
returns INTEGER
```

The found token's external code is determined by the value of the field `smCode` in the token structure (`smToken->smCode` in 'c') when the posthook function is terminated. This field is initially set to the value as specified in the vocabulary file or the vocabulary section depending on the token recognised.

The external token code for the token found can be changed in two ways, either by setting `smToken->smCode` to the new external code or by returning the new external code by executing a *return* statement. The external token code should be one of the enumeration values defined for tokens in the vocabulary to which the current token belongs, or the predefined enumeration `smSkipToken`.

When `smSkipToken` is returned the current token is skipped, as if it was given in the skip section.

scanner is used, the scanning is aborted and eventually the character is returned as an undefined token.

Any number of character sets can be defined in addition to the built-in allowing easy generation of the same scanner for different character sets.

3.2.12 The Map Section

```
<map section> ::=
     '%%MAP' {<character map>} ['%%END']

<character map> ::=
     <character class> '=' <character class> ';'
```

This section defines the mapping of a character read in the scanner. This can for example be used to map lower case characters on the upper case, creating a case-insensitive scanner (as opposed to character sets which handle the complete representation of characters and should be changed by changing the setting of the set option).

To define a character map character classes are used. Characters are given on the left-hand side and the equivalent characters on the right-hand side. For example:

```
[0-9] = [\xF0-\xF9];
```

maps the digits to the characters with hexadecimal values F0 and F9.

The characters in the character class are ordered from the lowest to the highest character value. The mapping is then performed by assigning the lowest character on the left-hand side to the value of the character on the right-hand side, and so on until the highest value is reached.

If the set of characters on the right-hand side contains fewer characters than that on the left-hand side the highest value of the set with the least number of characters is assigned to the remaining characters of the left-hand side. For example:

```
[0-9] = \x0;
```

maps all digits to the hexadecimal value 0 (zero).

If the left-hand side has fewer characters than the right-hand side the remaining characters are discarded. For example:

```
[0-9] = [A-Z];
```

maps all digits to the ASCII values `A` to `J` respectively.

If the character class begins with a character that is greater than the last character in the specified set of characters, the characters are processed in reversed order. For example:

This is further described in the description of semantic actions below.

3.2.14 The Vocabulary Section

```
<vocabulary section> ::=
     '%%VOCABULARY' <vocabulary name>
     {<token name> '=' <external token code> ';'}
     {<scanner section>}
```

The vocabulary section is used to specify the vocabulary used by the scanners defined for a specific vocabulary. A vocabulary also defines a set of tokens. The tokens can either be specified in a vocabulary file or directly after the name of the vocabulary in the vocabulary section. Each vocabulary may specify a number of scanners to recognise its set of tokens.

A token must always have an external token code which must be unique in the vocabulary. Tokens may be *string token*s, in which case the name of the token is given as a quoted string. All string tokens are automatically defined in the first scanner in the vocabulary that defines them if they are not explicitly defined in a scanner.

ScannerMaker will complain if a token have not been defined, or explicitly undefined, in any of the scanners defined for the vocabulary.

3.2.15 The Scanner Subsection

```
<scanner section> ::=
     '%%SCANNER' <scanner name> [':' <scanner name>]
     [<screened token section>]
     [<undefine token section>]
     {<rule definition section>}
```

where the rule definition section is

```
<rule definition section> ::=
       <rule section>
     | <skip section>
```

The scanner section defines a scanner. It consist of a scanner name, an optional screening section, an optional undefine token section, and rule definition sections. The name of the scanner is local for each vocabulary. That is, the same scanner name can be used in several vocabularies. However, a vocabulary may only define *one* scanner with a specific name.

The definition

```
<scanner name> ':' <scanner name>
```

defines a scanner which inherits definitions from another scanner. The new scanner copies all rules and semantic actions from the other scanner. For example:

```
%%SCANNER newScanner : oldScanner %%RULE
```

defines that the externally visible token integer is to be an integer, as defined in the *definition section* above, and it is also an integer followed by a lookahead string `'..'`. This is a rather perplexing definition because of the use of the token `integer` defined in the *definition section* in the regular expression and the definition of an external visible token which may be defined in the vocabulary file. However this example shows that it is possible to use the same name for a token defined in the *definition section* and a token defined in the *rules section* without name clash. The example also shows the possibility to give multiple internal definitions for the same externally visible token. That is, both definitions return the same external token code.

An alternative definition of integer could be

```
integer   = integer / '..'?;
```

which is an integer followed by an optional lookahead string `'..'`. Both definitions define exactly the same token but the first set of definitions creates a more efficient scanner because the lookahead is fixed to two characters while the other definition uses a variable length lookahead of either none or two characters.

> *Note:* If possible use fixed length regular expression either in the regular expression preceding the lookahead or in the lookahead (or both) when lookahead is used.

The scanner always tries to find the longest possible token, even when using lookahead. For example

```
absurd    = [0-9]+/[0-9]+;
```

locates a string which has one or more digits followed by at least one digit. However, such a definition is absurd because there is no definite way to determine when the lookahead starts but with the convention to always locate the longest token, even this type of definitions has a well defined meaning. The rule

```
absurd    = [0-9]+/[0-9];
```

will find the same token as above but is more efficient because the lookahead has a fixed length of one character.

3.2.19 The Skip Subsection

```
<skip section> ::=
     '%%SKIP' {<skip rule>} ['%%END']

<skip rule> ::=
       <token name> '=' <lookahead rule> [<action>] ';'
     | <string> '=' <lookahead rule> [<action>] ';'
```

This section defines the tokens that should be skipped by the scanner, that is, not be passed to the caller of the scanner function. For example

```
blank = [ \t\n];
```

```
| <regular expression> '{' '-' <m> '}'
| <regular expression> '{' <n> '-' <m> '}'
```

The first closure repeats the regular expression zero or more times, the second form repeats the regular expression one or more times while the third form means zero or one time. The other forms indicate a more general form where <n> is a number indicating the minimum number of times which the regular expression should be repeated, if missing zero is assumed. <m> is a number indicating the maximum number of times which the regular expression should be repeated, if missing infinite number of times is assumed. If only <m> is given a repetition of exactly <m> times is assumed.

The operations on regular expressions using curly braces should be used with care because it tends to create large state tables. For example

```
complex{6}
```

is equivalent to

```
complex complex complex complex complex complex
```

If `complex` derives 20 states the expression above will derive 120 states.

Cut

```
<cut> ::=
     <regular expression> '.'
```

When a token up to the cut operator is found the scanning is immediately abandoned. For example, this operator is very useful to describe comments in for example 'c'

```
comment = '/*' [^]* '*/'.;
```

The meaning of the rule above is to find a '/*' prefix and then match any character up to and including the first occurrence of a '*/'. With no cut operator the matched token would be up to the last occurrence '*/' in the input stream.

Another way to look at the example above is that the cut operator selects the shortest possible token which matches the definition. Without a cut operator the longest possible token which matches the definition will be selected.

Grouping

```
<grouping> ::=
     '(' <regular expression> ')'
```

Grouping are used to alter the priority of operations. For example:

```
('ab' ! 'cd')+
```

matches tokens with one or more occurrence of the 'ab' or 'cd' patterns.

Character String

```
<character string> ::=
     ''' { <character> } '''
```

A character string represents the regular expression needed to recognize that string. For example

```
'BEGIN'
```

is interpreted exactly as

```
[B][E][G][I][N]
```

Inside a character string the non-printable characters can be represented in the same way as non-printable characters in a character class. However the characters '^', '-', and ']' has no special meaning in a character string. To use a single quote, ''', inside a character string it must be preceded by a backslash ('\').

For example

```
'can\'t'
```

represents the character sequence

```
can't
```

Identifier

An identifier defined in the *definition section* can be used in regular expressions. In such a case the definition of the identifier is inserted in that place of the regular expression. For example:

```
%%DEFINITION
     integer   = [0-9]+;
%%RULE
     FIXATION  = integer[.]integer;
     FLOAT     = integer[.]integer([Ee][+\-]integer)?;
     INTEGER   = integer;
```

End of Text

```
<end of text> ::=
     '_EndOfText'
```

This special symbol matches the end of the input stream, the end of text. The end of the input stream is reached when the reader (as defined in the *reader section*, see *The Reader Section* on page 146) returns zero.

The `_EndOfText` symbol is case insensitive and must be the full regular expression. That is, `_EndOfText` can not be combined with selection, concatenation or closure.

Unknown

```
<unknown> ::=
     '_Unknown'
```

This special symbol matches all unknown tokens found in the input stream. An unknown token is always only one character long but may be manipulated as any other token.

The _Unknown symbol is case insensitive and must be the full regular expression. That is, _Unknown can not be combined with selection, concatenation or closure.

Operator Priorities and Associativity

In the table below each operator is given in order of priority, and with its associativity:

Priority	Operator	Name	Type	Associativity
1	.	cut	unary	none
2	*	closure	unary	none
2	+	closure	unary	none
2	?	closure	unary	none
2	{ ... }	closure	unary	none
3		concat.	binary	left
4	!	selection	binary	left

The highest operator priority is 1 and lowest priority is 4.

3.2.21 Semantic Actions

```
<action> ::=
     '%%' <any character sequence except '%%'> '%%'
|    '%%DO' <action name>
```

In the *rule* and *skip sections* an action can be placed after each token definition, before its trailing semicolon, ';'. The action begins and ends with two percent characters, '%%', or a reference to a semantic action defined in the *definition section* using the '%%DO' keyword.

All characters inside an action are considered to be code written in the same language as the scanner skeleton. This language is the same as the language specified in the TARGET option. The code in a semantic action is executed when the corresponding token is found. For example:

```
integer   = digit+
%%
     unsigned char tmp;

     tmp = smThis->smText[smThis->smLength];
     smThis->smText[smThis->smLength] = 0;
     smToken->ival = (int)atoi(smThis->smText);
     smThis->smText[smThis->smLength] = tmp;
%% ;
```

Definitions from the *definition section* can be referenced. For example:

```
Integer = Integer %%DO Copy;
```

Using the definitions above in the description of the *definition section* the integer found is copied to `aBuffer`.

Similarly, target language code can be specified in the *declaration*, *context*, *reader*, *action*, *prehook* and *posthook sections*. Each of these target language code sections begins with its keyword and ends with anything starting with two percent characters. For example:

```
%%DECLARATION
     int commentLevel=0;
     int commentModifier=0;
%%MAP
     ...
```

By default the character '`' has special meaning in the target language code sections. This character is called the escape character (see *The Escape Directive* in the *ToolMaker System Description*, page 32). By giving an escape character the following character is unconditionally processed. For example, if double percent characters should be used in the code they should be preceded by the escape character. For example

```
%% printf("%d`%%\n",percent); %%
```

is translated to

```
printf("%d%%\n",percent);
```

instead of terminating at the first pair of percent characters. The escape character is escaped by itself. Thus, '` `' means '`'.

The code in a semantic action is executed when a token that matches the corresponding definition is found. Each semantic action may be viewed as a function with the following definition:

```
code = smScAction(smThis,smInternalCode,smToken)

smThis     : IN smScContext
smToken    : IN OUT %%(tokenType)
returns INTEGER
```

In addition the code given in the *action section* is executed first. The token's external code is determined by the value of the field `smCode` in the token structure (`smToken->smCode` in 'c') when the function is terminated. This variable is initially set to the value as specified in the vocabulary file or the vocabulary section or as set in a possible *action section* (see *The Action Section* on page 146).

The external token code for the token found can be changed (simulating finding of another token) in two ways by either setting `smToken->smCode` to the new external code or by returning the new external code by executing a *return* statement. The external token codes should be one of the enumeration

values defined for tokens in the vocabulary to which the current token belongs, or the predefined enumerators `smSkipToken` or `smContinueToken`.

When `smSkipToken` is returned the current token is skipped, as if it had been specified in the skip section.

When `smContinueToken` is returned continued scanning will be performed, *Continued Scanning* on page 171.

3.3 The ToolMaker Common Description file

Unless ScannerMaker is the only Maker used, common declarations of the source position and the token structures should be placed in the ToolMaker Common Description file which is described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. Otherwise these two sections may be specified in the ScannerMaker Description file, removing any need for the ToolMaker Common Description file.

3.4 The Vocabulary file

The vocabulary file used by ScannerMaker corresponds to the vocabulary file produced by ParserMaker. The format of the vocabulary file consist of four fields:

1) The number of the row in the vocabulary file.

2) The external token code number.

3) The token name. The token name may either be an identifier; a letter followed by zero or more letters, digits or underscores, or a string as defined by ScannerMaker.

4) The name of the vocabulary to which the token belongs. The vocabulary name should be an identifier; a letter followed by zero or more letters, digits or underscores.

Each field should be separated by blank characters. If multiple vocabularies are defined they must all be defined in the same vocabulary file, ScannerMaker only reads one vocabulary file. Below is an example of a vocabulary file:

```
0    0    Unknown        main
1    1    EndOfText      main
2    2    ';'            main
3    3    '('            main
4    4    ')'            main
5    5    ','            main
```

Each token should appear only once for each vocabulary and two token may not have the same external token in the same vocabulary.

For compatibility an older format is also supported. The format of the older vocabulary file is not described. However, if this format is used the end-of-text marker, '`$`', is replaced with the `EndOfText` token and all tokens are defined to belong to a scanner called `main`.

4 THE SCANNERMAKER COMMAND

The invocation of the scanner generator is performed by the following program call

```
smk [-help] [<options> ...] <file>
```

The special option `-help` gives a short help of available options and their meaning.

The default extension on the ScannerMaker description file is **.smk** which is added if not explicitly stated.

4.1 Parameters

The command takes only one argument, the name of a description file. The description file contains a description of the scanner to be made using a formal language as described in *The ScannerMaker Description File* on page 138. The argument may appear anywhere on the command line except between any option and its argument.

4.2 Options

One or more options may be specified on the command line overriding any specified options in the *options section* in the description file. The only options which can not be specified in the *options section* are

```
Option
-help
-voc <file>
-smt <file>
-sml <file>
```

All other options have the same names as in the *options section*. For a detailed description on how to specify options on the command line see *Command Line Option Format* in the *ToolMaker System Description*, page 40.

The following options have no corresponding directive in the *options section* of the description file and are therefore thoroughly described:

```
-help
```

This special option produce a verbose listing of the usage format of the command. Each argument and option is given a short explanation.

```
-voc <file>
```

The vocabulary used is read from the `<file>` as specified as argument to this option. By default the vocabulary is read from a file with the same base name as the description file but with extension **.voc**.

```
-sml <file>
```

The specified file is used as list file. By default the name of the list file is the same as the description file but with extension **.sml**. Note that this option does not produce any list file if no list directives are used.

```
-smt <file>
```

The specified file is used as the table file. By default the name of the table file is the same as the description file but with extension **.smt**. Note that this option does not imply that the table file should be saved after source code generation, it simply indicates its name (see also *The Maker* in the *ToolMaker System Description*, page 25 for a description of the intelligent generation strategy and *File Descriptions* on page 138).

5 SCANNER RUN-TIME USAGE

5.1 Principles of Operation

To use a ScannerMaker generated scanner the following step should be performed:

- create a new scanner context (see below)
- initialise the reading from the input stream (e.g. open the file and attach it to the context)
- call ***sm***Scan until end of input

During each call to the scanner the appropriate semantic action will automatically be performed (see *Semantic Actions* on page 158) in which various of the features of the generated scanner can be used. It is also possible to create additional contexts, using the function ***sm***ScNew, and switch scanner and vocabulary during the scanning process.

5.1.1 Type: ***sm***ScContext and ***sm***ScContextItem

```
TYPE smScContext IS POINTER TO smScContextItem

TYPE smScContextItem IS STRUCTURE
     smSize     : INTEGER
     smText     : STRING
     smLength   : INTEGER
     smBufferOverflow: INTEGER
     smPosition: INTEGER
     smLine     : INTEGER
     smColumn   : INTEGER
     smNextPosition: INTEGER
     smNextLine: INTEGER
     smNextColumn: INTEGR
     smScanner : smScScanner

          declarations from the context section

END STRUCTURE
```

A context is the main object in the generated scanner. Almost all functions operate on that object. A context consist of a buffer, source position, scanner, and user defined variables.

In object oriented languages this type is declared as a class named ***sm***ScContext with the structure components above as public members.

> *Note:* All predefined variables starts with the characters sm and can not be changed. This is *not* the scanner prefix used as a prefix for the exported functions and the context data type.

The Context Buffer

```
smSize    : INTEGER
```

The context buffer is divided into two parts, the token and the input stream, where the input stream always follows the token. Initially the token length is zero and when end of text is found the input stream has a length of zero. The size of the context buffer can be found in the `smSize` field. Initially the length of the context buffer is zero.

```
smText    : STRING
smLength  : INTEGER
```

The variable `smText` holds the last scanned token and `smLength` is the length of the token. The characters in `smText` may be altered freely but if any altered characters are skipped back into the input stream the source position calculation will be wrong if any newline characters was altered or inserted.

Great care should be taken not to alter any characters after the last character in the token permanently. It is alright to alter the character while in a semantic action and then restoring the old character before leaving the semantic action or calling the scanner recursively. For example, to convert a token representing an integer into an integer, the unix standard function `atoi` could be used. However, this function requires that the string representation is null terminated but ScannerMaker does not null terminate a token string so this must be done explicitly.

```
Integer = [0-9]+
%%
{
     unsigned char *remind;

     remind = smThis->smText[smThis->smLength];
     smThis->smText[smThis->smLength] = 0;
     smToken->ival = atoi(smThis->smText);
     smThis->smText[smThis->smLength] = remind;
}
%%;
```

It is very important to restore the character over-written by the null character as in the example above or else the next token would start with a null character.

```
smBufferOverflow: INTEGER
```

If the scanning of a token exceeds the maximum token length or the reallocation to a larger buffer fails, characters in the buffer will be skipped. The number of characters saved equals to the minimum length of the token and the only characters saved are the last one in the buffer. The variable `smBufferOverflow` holds the number of characters which is skipped. The source position will still however reference the true beginning of the character but the `smLength` will not include any skipped characters. Thus the true length of the token is the sum of `smBufferOverflow` and `smLength`.

If no overflow occurs `smBufferOverflow` will be zero.

Changing the context can be necessary when changing from one source of input to another. This can be achieved by simply replacing the current context variable with a new (created using the **sm**New function). To be able to restore the previous context the context buffers should be linked or stacked, which is the responsibility of the user

The change of a context should preferably be performed *outside* the actual generated scanner, e.g. in a function enclosing the **sm**Scan function, which recognises and handles situations when this is necessary.

> *Note:* The *prehook*, *action* and *posthook section* and the semantic actions are all called using the same context (the one passed to **sm**Scan).

Source Position

```
smPosition: INTEGER
smLine    : INTEGER
smColumn  : INTEGER
```

The source position is only calculated if the appropriate source position directive is given in the *source position section*. The line number and column number start from one and the position from zero. The position is the number of characters read in this context before the token. The source position will always be correct with one exception. If altered characters are returned to the input stream by using **sm**ScSkip instead of **sm**ScModify and any newline is altered to a character other than the newline character or if a character is altered to a newline, then the source position will be computed incorrectly. However, if the number of newlines are the same the source position will be correct after the next newline following the altered characters. The suggestion is not to return any altered character with the **sm**ScSkip function.

When **sm**ScModify is used to return altered characters to the input stream any token found containing altered characters will have the source position of the token following the token which altered the input stream. For example

```
'hello' = 'hello' %% smScModify(smThis,"theis",-5) %%;
'the' = 'the' %% ... %%;
'is' = 'is' %% ... %%;
```

when the token 'hello' is found, the token is altered to 'theis'. If the source position of 'hello' was line one and column five, then the next token should be found at line one and column ten. This is the line number which is reported by 'the', 'is', and the token following 'is'.

```
smNextPosition: INTEGER
smNextLine    : INTEGER
smNextColumn  : INTEGER
```

The next positions gives the next token's position. That is, the token following the current one.

Scanner

```
smScanner : smScScanner
```

Which scanner a context should use is determined by the value of the smScanner variable. Available scanners are determined by the description file defining the scanners. An enumeration type is defined containing an enumeration literal for each scanner. The name of the enumeration literal is the scanner prefix concatenated by the vocabulary name and scanner name surrounded by underscores and finally the characters 'Scanner' as a suffix. For example, if the scanners main and comment are defined the following enumeration type is defined (if 'c' is the target language):

```
typedef enum smScScanner {
     sm_MAIN_MAIN_Scanner= 0,
     sm_MAIN_COMMENT_Scanner= 1,
     sm_SET_MAIN_Scanner = 2
} smScScanner;
```

By simply changing the value of the smScanner field in the context variable a new scanner is entered. The variable can be changed any time and the effect is immediate. For example, it is possible to change scanner in a semantic action and then continue scanning using the newly selected scanner.

For each vocabulary there is also an enumeration type defined containing enumeration literals for all tokens defined in that vocabulary. The name of the enumeration type is *sm* followed by the vocabulary name enclosed by underscores and followed by the characters 'ScToken'. Each enumeration literal is named *sm* followed by underscore, vocabulary name, underscore, token name or external token code for a string token, underscore, and Token. For example, if 'c' is the target language:

```
typedef enum sm_main_ScToken {
     sm_MAIN_UNKNOWN_Token   = 0,
     sm_MAIN_ENDOFTEXT_Token = 1,
     sm_MAIN_2_Token         = 2,
     sm_MAIN_INTEGER_Token   = 3,
          :
          :
} sm_main_ScToken;
```

5.2 Run Time Interface

The run time interface consist of a number of functions. All functions except smScScrScanner, smScScrToken, and smScScrRule operates on a context. In a non-object oriented language the context is always given as the first argument. In an object oriented language these functions are member functions to the smScContext class and messages are sent to a context object.

5.2.1 Function: smScan

```
code = smScan(smThis,smToken)
```

```
smThis    : IN smScContext
smToken   : IN OUT %%(tokenType)
returns INTEGER
```

This function scans a new token. If successful, the token code is returned and the token structure or record is assigned the token code and source position, if used. If the reader, as defined in the reader section, returns a negative value this is immediately propagated as the return value of the smScan and the scanning is aborted.

5.2.2 Function: smScNew

```
context = smScNew(smScanner)

smScanner : IN smScScanner
returns smScContext
```

This function is a so called constructor. It constructs a new context with the specified smScanner.

5.2.3 Function: smScDelete

```
smScDelete(smThis)

smThis    : IN smScContext
```

This function is a so called destructor. It destroys a context and its buffer.

5.2.4 Function: smScSkip

```
length = smScSkip(smThis,smLength)

smThis    : IN smContext
smLength  : IN INTEGER
returns INTEGER
```

This function changes the token by either replacing characters from the end of the token to the input stream, or by extending it by reading characters from the input stream. The number of characters replaced or read are given by the value of smLength. A negative value replaces characters and a positive value reads characters. The actual number of characters replaced or read are returned. For example,

```
smScSkip(smThis,-3);
```

returns the three last characters of the token to the beginning of the input stream. The length of the token will be three characters shorter. However, if it has a length of less than three characters, for example one character, only that many characters will be returned. If the reader, as defined in the reader section, needs to be called to fetch further characters from the input stream and it returns a negative value, this is immediately propagated as the return value of smScSkip and the skipping is terminated.

5.2.5 Function: *sm*ScModify

```
length = smScModify(smThis,smBuffer,smLength)

smThis    : IN smScContext
smBuffer  : IN STRING
smLength  : IN INTEGER
returns INTEGER
```

This function modifies the token or input stream in the same way as *sm*ScSkip but all affected characters (replaced to the input stream or appended to the token) are modified by characters from the specified buffer. For example

```
smScModify(smThis,"123",3);
```

appends the token with three characters from the input stream and modifies them to be "123". That is, the token now ends with "123".

```
smScModify(smThis,"123",-3);
```

returns the three last characters of the token to the beginning of the input stream and replaces them with "123". If the reader, as defined in the reader section, returns a negative value this is immediately propagated as the return value of *sm*ScModify and the modification is terminated.

5.2.6 Function: *sm*ScCopy

```
length = smScCopy(smThis,smBuffer,smFrom,smTo)

smThis    : IN smScContext
smBuffer  : OUT STRING
smFrom    : IN INTEGER
smTo      : IN INTEGER
returns INTEGER
```

Copy the token into the buffer specified. The token's characters are *not* mapped. The copy starts to copy characters from the smFrom position up to but not including the smTo position from the current token. The number of characters copied is returned. For example, to copy all characters use:

```
smScCopy(smThis,buffer,0,smThis->smLength);
```

5.2.7 Function: *sm*ScMapCopy

```
length = smScMapCopy(smThis,smBuffer,smFrom,smTo)

smThis    : IN smScContext
smBuffer  : OUT STRING
smFrom    : IN INTEGER
smTo      : IN INTEGER
returns INTEGER
```

Copy the token into the buffer specified in the function call. The characters of the token are mapped using the specified character map as specified in the *map section*. The copy starts to copy characters from the smFrom position up

to but not including the smTo position from the current token. The number of characters copied is returned.

5.2.8 Function: *sm*ScScrScanner

```
void smScScrScanner(smScanner,smOnOff)

smScanner       : IN smScScanner
smOnOff         : IN BOOLEAN
```

This function turns on or off checking of screened tokens for the specified scanner. If the scanner has screened tokens these are not recognized if screening is turned off.

5.2.9 Function: *sm*ScScrToken

```
void smScScrToken(smScanner,smToken,smOnOff)

smScanner       : IN smScScanner
smToken         : IN sm_vocabularyname_ScToken
smOnOff         : IN BOOLEAN
```

This function turns on or off checking of screened tokens for the specified token. If the token has screened tokens these are not recognized if screening is turned off.

5.2.10 Function: *sm*ScScrRule

```
void smScScrRule(smScanner,smToken,smOnOff)

smScanner       : IN smScScanner
smToken         : IN sm_vocabularyname_ScToken
smOnOff         : IN BOOLEAN
```

This function turns on or off checking of the specified token. If the token is a screened token it is not recognized if screening is turned off.

5.3 Recursive Calls and Continued Scanning

There are two ways in which scanning may proceed directly from inside a semantic action without actually returning from the current call. Either the scanner can be called recursively or the scanning may be continued.

5.3.1 Recursive Calls

Inside any action or target language code section it is possible to call the scanner recursively. However smToken should *never* be passed to any called scanner because it will over-write the current token code. Also note that the variable smToken is not defined in the *code section*.

The current token is reset by a recursive call. That is, after a recursive call the smText and smLength will refer to the last scanner token. For example

```
INCLUDE = '#include'
%%
     TmToken myToken;

     /* #include "astring" */

     if(smScan(smThis,&myToken)==2){

          /* open file and create a new context */

     } else {
          error("Illegal include statement");
     }
%%
```

After calling ***sm***`Scan()` the string `'#include'` which could be found in `smText` is replaced by the next token, in this case hopefully a string.

5.3.2 Continued Scanning

It is also possible to continue scanning after a token is found by setting the external code to `smContinueToken` or returning `smContinueToken` in the action section or the semantic actions. The current token will be appended with the new token found after a continued scanning is made. The new token will be returned.That is, in the example below the complete string will be found in the last semantic action.

For example

```
STRING = '"' [^"] '"' / '"'
          %% return smContinueToken; %%;
STRING = '"' [^"] '"'
          %% ... %%;
```

or

```
STRING = '"' [^"] '"' / '"'
          %% smCode=smContinueToken; %%;
STRING = '"' [^"] '"'
          %% ... %%;
```

Continued scanning can only be used in the *action section* and inside semantic actions.

A THE PL/0 EXAMPLE

This example is a part of the pl/0 example used throughout the ToolMaker documentation. The *ToolMaker System Description* contains a detailed walk-through of the relevant parts of the example. This appendix contains the files relevant for ScannerMaker.

A.1 pl0.smk - The ScannerMaker Description File

```
------------------------------------------------------
-- pl0.smk     Date: 1993-06-24/toolmake
--
-- pl0 - ScannerMaker description file
--
------------------------------------------------------
-- Created:    1993-04-27/reibert@roo
-- Generated: 1993-06-24 12:42:46/toolmake v2,r0,c12
------------------------------------------------------

%%OPTIONS
     Verbose;

%%IMPORT
#include <stdio.h>

#include "pl0List.h"
#include "pl0Scan.h"

%%EXPORT

extern void scan(TmToken *token);
extern int pl0ScanEnter(char *fileName);
extern void pl0ScanTerminate(void);

%%CONTEXT
  char *fileName;
  int fd;
  pl0ScContext previous;
  int fileNo;

%%READER
    return read(smThis->fd, (char *)smBuffer, smLength);

%%DECLARATIONS

static pl0ScContext lexContext;

static int fileNo = 0;/* Count included files */

int pl0ScanEnter(
     char fnm[]/* IN - Name of file to open */
){
  pl0ScContext tmp;

  tmp = pl0ScNew(pl0_MAIN_MAIN_Scanner);
  if (fnm == NULL)
    tmp->fd = 0;
  else if ((tmp->fd = open(fnm,0)) < 0) {
    pl0ScDelete(tmp);
    return 0;
```

```
  } else {
    tmp->fileName = fnm;
    tmp->fileNo = fileNo++;
    tmp->previous = lexContext;
    lexContext = tmp;
  }
  return 1;
}

/*-------------------------------------------------------

  pl0ScanExit()

  Terminate and delete the current lexical context.

*/
static void pl0ScanExit(void)
{
  pl0ScContext old;

  close(lexContext->fd);
  old = lexContext;
  lexContext = lexContext->previous;
  pl0ScDelete(old);
}

/*-------------------------------------------------------

  pl0ScanTerminate()

  Make sure all contexts are terminated (e.g. in case of
  parser abort).

*/
void pl0ScanTerminate(void)
{
  while (lexContext)
    pl0ScanExit();
}


/*-------------------------------------------------------

  scan()

  Outer scanner called from the parser. Handles switching
  of contexts.

*/
void scan(
  TmToken *token
){
  pl0Scan(lexContext, token);

  switch (token->code) {

  case pl0_MAIN_INCLUDE_Token:
    if (pl0ScanEnter(token->stringValue)) {
      TmSrcp srcp, start;

      srcp = token->srcp;
      srcp.line++;/* Make include start on next line */
      srcp.col = 1;
```

```
      start.file = fileNo;
      start.line = 0;
      pl0LiEnter(&srcp, &start, token->stringValue);
    } else {
        pl0Log(&token->srcp, 199, sevFAT,
              token->stringValue);
    }
    /* Get next token and return instead of the INCLUDE */
    pl0Scan(lexContext, token);
    break;

  case pl0_MAIN_ENDOFTEXT_Token:
    pl0ScanExit();
    if (lexContext)/* If still more input get a token */
      pl0Scan(lexContext, token);
    break;
  }
}

%%POSTHOOK

  smToken->srcp.file = smThis->fileNo;

%%DEFINITIONS

  Letter = [A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\xFF];
  Digit  = [0-9];
  White  = [ \t\n];

%%VOCABULARY main

%%SCANNER main %%RULES

  NUMBER = Digit+
     %%
       pl0ScCopy(smThis,
              (unsigned char *)smToken->stringValue, 0,
              smThis->smLength);
       smToken->stringValue[smThis->smLength] = 0;
       smToken->integerValue=atoi(smToken->stringValue);
     %%;

  IDENTIFIER = Letter ('_' ? Letter ! '_' ? Digit)*
     %%
       pl0ScCopy(smThis,
              (unsigned char *)smToken->stringValue, 0,
              smThis->smLength);
       smToken->stringValue[smThis->smLength] = 0;
     %%;

  Include = '$INCLUDE'
     %%
       TmToken token;
       int i;
       char c;

       pl0Scan(smThis, &token);    /* Get file name */
       do {
        i = pl0ScSkip(smThis, 1);
        c = smThis->smText[smThis->smLength-1];
       } while(c != '\n' && i != 0); /* Skip to EOL or EOF */
        strcpy(smToken->stringValue, token.stringValue);
     %%;
```

```
  Unknown = _Unknown;
  EndOfText = _EndOfText;

%%SKIP

  Blank = White+;              -- Skip any white space
  Comment = '--' [^\n]*[\n];-- and Ada style comments

%%END
```

B ERROR MESSAGES

Error messages produced by ScannerMaker are output to the terminal, and if the option `List Input;` is set, messages are also output to the list file.

For a description of the format of error messages refer to *Message Format* in the *ToolMaker System Description*, page 49.

B.1 Message Explanations

The following list gives a brief summary of the error messages and in some cases the actions that should be taken by the user. An error message containing '`%n`', where `n` is a number, means that an insert string is inserted in that position during the actual generation of the message.

Messages with numbers less than 100 are messages common for all Makers. These are described in *Messages Explanations* in the *ToolMaker System Description*, page 50. Messages indicating license problems or limitations are described in *License Errors* in the *ToolMaker System Description*, page 51.

100 `Parsing resumed.`

This error specifies a point in the description file where parsing could be resumed after a wild error.

101 `%1 inserted.`

The token or tokens is inserted into the description file because the ScannerMaker thought it was missing.

102 `%1 deleted.`

The token or tokens is deleted from the description file because the parser thought it was wrongly inserted.

103 `%1 replaced by %2.`

The token is replaced with another token. More than one token can be replaced with one or more tokens.

104 `syntax error, stack backed up.`

Severe syntax error there the normal error recovery strategies failed and the parser was forced to do severe error recovery. The meaning of the scanner description is probably lost.

105 `Syntax error.`

An unspecified syntax error. This error message should not appear.

106 `Parser stack overflow`

107 `Parse table error.`

108 `Parsing terminated.`

These are errors in the scanner generator. Contact your ToolMaker contact person for bug report.

207 `Can not overwrite table file "%1".`

The protection of an existing table file prevents ScannerMaker to over-write it.

208 `Can not overwrite list file "%1".`

The protection of an existing list file prevents ScannerMaker to over-write it.

209 `Scanner "%1" already defined.`

The scanner is already defined earlier in the description file.

210 `Token "%1" already screened.`

The token is already screened in this scanner.

211 `Token "%1" not in vocabulary.`

The token can not be found in the vocabulary.

212 `External code already exist.`

The external token code is already used in this scanner.

213 `Token name already exist.`

The token is already defined in this scanner.

214 `Multiple matches of END OF TEXT token in scanner.`

Only one token can be defined to match the end of text.

215 `Multiple matches of UNKNOWN token in scanner.`

Only one token can be defined to match unknown tokens.

216 `END OF TEXT token not defined.`

There is no token which match the end of text.

218 `UNKNOWN token not defined.`

There is no token which match unknown tokens.

220 `String contains mapped or excluded character, or characters not in the current character set.`

Only characters in the current character set are allowed within string tokens. Mapped or excluded characters are not allowed.

221 `Automatically defined in "%1".`

The string token is not defined in the scanner but it is automatically defined to match itself.

222 `Not defined in "%1".`

The token is not defined in the scanner.

223 `Vocabulary file "%1" not found.`

The vocabulary file can not be found.

224 `Line number expected.`

Line number is expected in the first field of the vocabulary file.

225 `External code expected.`

External token code is expected in the second field of the vocabulary file.

226 `Old end of text symbol, converting to END OF TEXT.`

The old end of text symbol `'$'` is found. Convert it to the token `EndOfText`.

227 `Token name expected.`

Token name is expected in the third field of the vocabulary file.

228 `Old format, using "main" as vocabulary.`

Old format of vocabulary file. All tokens are considered to belong to the vocabulary `main`.

229 `Scanner name expected.`

Scanner name is expected in the fourth field of the vocabulary file.

230 `End of line expected.`

End of line is expected in the vocabulary file.

231 `Must be one character long.`

The character string must be one character long. The character may be quote character.

232 `Set not uniquely mapped.`

The set is not unique. That is, each character in the set must have a unique character equivalent or be skipped.

233 `Token already defined.`

Token is already defined.

234 `Scanner not found.`

The scanner reference is not found.

235 `Token reference not found in scanner "%1".`

The token referenced in the scanner is not found.

236 `Token not defined.`

The token is not defined in the scanner.

237 `Number must be greater than or equal to the first number.`

The first number in the general closure specification must be less than the last number.

238 `Class contain mapped or excluded characters.`

Mapped or excluded characters are not allowed in a class.

239 `Token not defined.`

The token is not defined in the definition section.

240 `Scanner description file "%1" not found.`

The description file can not be found.

241 `The cut operator must lead to a final state.`

The cut operator must be used in such as way that the character preceding it matches the last character in the regular expression.

242 `Not accepted by "%1".`

The screened token is not accepted by the token used to screen it.

243 `%1 definition takes precedence (ambigous accepting state).`

Two or more tokens have the same accepting state.

244 `Old format, defines UNKNOWN with external code 0.`

Old format of the vocabulary file implicitly defined unknown tokens to be 0.

245 `Action not defined.`

The action is not defined in the definition section.

246 `Action already defined.`

An action can only be defined once.

247 `Name of set expected.`

The name of the set being defined was expected.

248 `Set already defined.`

A set with the same name is already defined.

249 `Unexpected end of set.`

More characters are expected in the set.

250 `Character already defined.`

It is illegal to define the same character twice in a set.

251 `Unknown token, ignored.`

Unknown token found in the set.

252 `Character %1 multiply defined in %2 set.`

It is illegal to define the same character twice in a set.

254 `Specified set %1 does not exist.`

The set is not defined.

255 `Vocabulary already defined.`

Vocabularies must have unique name. A vocabulary with the same name has already been used in this description file.

256 `Vocabulary not defined.`

The vocabulary can not be found in the vocabulary file.

257 `Scanner already defined.`

A scanner with the same name is already defined in the same vocabulary.

258 `Scanner not defined.`

The scanner is not defined in the same vocabulary.

259 `Token is not defined in this vocabulary.`

The token must be defined in this vocabulary or be specified in the undefine section of the scanner.

260 `Screening turned off, no other token may screen it.`

No token was found in the scanner which could be used to screen the token. The token will be part of the scanner.

261 `Token predefined in %1 scanner.`

The token is predefined in the first scanner defined for the vocabulary.

262 `Section already defined.`

The specified code section can only be used once in the description file.

401 `Scanner description file "%1" not found.`

The scanner description file could not be found.

402 `Trouble producing code.`

It was not possible to generate any target language code. The problem may be that generated files could not be written due to protection problem, older protected files or protected current directory, or an error in the skeleton file.

405 `Target language "%1" not officially supported.`

This warning is issued if the target language is set to a language not officially supported. Note that the option value is case sensitive and the message may be caused by a typing error.

C DESCRIPTION LANGUAGE

This appendix contains a syntax summary of the ScannerMaker description language. For a full discussion of the language, see *The ScannerMaker Description File* on page 138.

For a brief discussion of the notation used in this description see appendix C in the *ToolMaker System Description, SYNTAX NOTATION*, page 52.

C.1 Language Syntax

```
<description file> ::=
          {<toolmaker section>}
          {<target code section>}
          {<set definition section>}
          {<general definition section>}
          {<vocabulary section>}

<toolmaker sections> ::=
     [ <options section> ]
     { <import section> | <srcp section>
          | <token section> }

<import section> ::=
     see the ToolMaker System Description

<srcp section> ::=
     see the ToolMaker System Description

<token section> ::=
     see the ToolMaker System Description

<options section> ::=
     '%%OPTIONS' {<directive>} ['%%END']

<directive> ::=
        <common directive>
      | <optimize directive>
      | <trace direvtive>
      | <set directive>
      | <pack directive>
      | <screening directive>
      | <list directive>
      | <token size directive>
      | <token limit directive>
      | <exclude character directive>

<common directive> ::=
        <target directive>
      | <os directive>
      | <prefix directive>
      | <library directive>
      | <escape directive>
      | <width directive>
      | <height directive>
      | <generate directive>
      | <force directive>

<optimize directive> ::=
     ['NO'] 'OPTIMIZE' ';'
```

```
<generate directive> ::=
     ['NO'] 'GENERATE' <generate> ';'

<generate> ::=
       'SOURCE'
     | 'TABLES'

<verbose directive> ::=
     ['NO'] 'VERBOSE' ';'

<trace directive> ::=
     ['NO'] 'TRACE' ';'

<set directive> ::=
     'SET' <set name> ';'

<set name> ::=
     <string>

<pack directive> :=
       'NO' 'PACK' ';'
     | 'PACK' <pack> {',' <pack>} ';'

<pack> ::=
       'ROW'
     | 'COL'
     | 'LES'
     | 'GCS'
     | 'RDS'
     | 'ERROR'

<screening directive>
       'NO' 'SCREENING' ';'
     | 'SCREENING' <minimum token size> ';'

<minimum token size> ::=
     <number>

<list directive> ::=
       'NO' 'LIST' ';'
     | 'LIST' <list> {',' <list>} ';'

<list> ::=
       'SCREENING'
     | 'TOKEN'
     | 'MAP'
     | 'SET'
     | 'NFA'
     | 'DFA'
     | 'RULE'

<escape directive> ::=
       'NO' 'ESCAPE' ';'
     | 'ESCAPE' <escape character>';'

<width directive> ::=
       'NO' 'WIDTH' ';'
     | 'WIDTH' <width> ';'

<width> ::=
     <number>
```

```
<height directive> ::=
      'NO' 'HEIGHT' ';'
    | 'HEIGHT' <height> ';'

<height> ::=
    <number>

<target operating system directive> ::=
    'OS' <operating system> ';'

<operating system> ::=
    <string>

<target language directive> ::=
    'TARGET' <language> ';'

<language> ::=
    <string>

<library directive> ::=
    'LIBRARY' <library> ';'

<library> ::=
    <string>

<prefix>
    'PREFIX' <system name prefix> ';'

<system name prefix> ::=
    <string>

<token size directive> ::=
    'TOKENSIZE' <minimal token size> ' ';'

<token limit directive> ::=
    'TOKENLIMIT' <maximal token size> ' ';'

<exclude character directive> ::=
    'EXCLUDE' <exclude character> ';'

<target code sections> ::=
      <declaration section>
    | <context section>
    | <export section>
    | <code section>
    | <reader section>
    | <prehook section>
    | <posthook section>
    | <action section>

<declaration section> ::=
    '%%DECLARATION' <target language code> ['%%END']

<context section> ::=
    '%%CONTEXT' <target language code> ['%%END']

<code section> ::=
    '%%CODE' <target language code> ['%%END']

<export section> ::=
    '%%EXPORT' <target language code> ['%%END']
```

```
<reader section> ::=
     '%%READER' <target language code> ['%%END']

<action section> ::=
     '%%ACTION' <target language code> ['%%END']

<prehook section> ::=
     '%%PREHOOK' <target language code> ['%%END']

<posthook section> ::=
     '%%POSTHOOK' <target language code> ['%%END']

<general definition section> ::=
       <map definition section>
     | <definition section>

<set section> ::=
     '%%SET' <set name> {<set>} ['%%END']

<set> ::=
       <character>
     | <hex digit> <hex digit>
     | '..'

<map section> ::=
     '%%MAP' {<character map>} ['%%END']

<character map> ::=
     <character class> '=' <character class> ';'

<definition section> ::=
     '%%DEFINITION' {<definition>} ['%%END']

<definition> ::=
     <definition name> '='
          [<selection rule>] [<action>]';'


<vocabulary section> ::=
     '%%VOCABULARY' <vocabulary name>
     <token name> '=' <external token code> ';'}
     {<scanner section>}


<scanner section> ::=
     '%%SCANNER' <scanner name> [':' <scanner name>]
     [<screened token section>]
     [<undefine token section>]
     {<rule definition section>}

<screened token section> ::=
     '%%SCREENING'
     {<token name> ';'}

<undefined token section> ::=
     '%%UNDEFINE'
     {<token name> ';'}
```

```
<rule definition section> ::=
      <rule section>
    | <skip section>

<rule section> ::=
    '%%RULE' {<token rule>} ['%%END']

<token rule> ::=
      <token name> '=' <lookahead rule> [<action>] ';'
    | <string>     '=' <lookahead rule> [<action>] ';'


<skip section> ::=
      '%%SKIP' {<skip rule>} ['%%END']

<skip rule> ::=
      <token name> '=' <lookahead rule> [<action>] ';'
    | <string>     '=' <lookahead rule> [<action>] ';'


<lookahead rule> ::=
      <selection rule> ['/' <selection rule>]

<selection rule> ::=
    [<selection rule> '!'] <concatenation rule>

<concatenation rule> ::=
    [<concatenation rule>] <closure rule>

<closure rule> ::=
      <item> '*'
    | <item> '+'
    | <item> '?'
    | <item> '{' <number> '}'
    | <item> '{' [<number>] '-' [<number>] '}'

<item> ::=
      <definition name>
    | <string>
    | <character class>
    | '.'
    | '(' <selection rule> ')'

<character class> ::=
      '['     {<character>} ']'
    | '[' '^' {<character>} ']'

<action> ::=
    '%%' <target language code> '%%'
```

Note that every keyword starting with '%%' can be specified using both upper or lower case letters and it may also be specified in plural.

C.2 Lexical Items

```
<upper case letter> ::=
      'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
    | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P'
```

```
    | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X'
    | 'Y' | 'Z'

<lower case letter> ::=
      'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
    | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
    | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x'
    | 'y' | 'z'

<letter> ::=
    <upper case letter> | <lower case letter>

<digit> ::=
      '0' | '1' | '2' | '3' | '4'
    | '5' | '6' | '7' | '8' | '9'

<special character> ::=
      '!' | '@' | '#' | '$' | '%' | '^' | '&' | '*'
    | '(' | ')' | '_' | '-' | '+' | '=' | '|' | '{'
    | '}' | '[' | ']' | ';' | ':' | ',' | '"' | '~'
    | '`' | '<' | '>' | ',' | '.' | '?' | '/' | '\'

<target language code> ::=
any characters in the target language except the escape
character, see The ScannerMaker Description File on page
138.

<token name> ::=
    <letter> {<letter> | <digit> | '_'}

<definition name> ::=
    <letter> {<letter> | <digit> | '_'}

<string> ::=
    ''' {<letter> | <digit> | <special character>} '''
```

D TARGET LANGUAGE DETAILS

This appendix describes the actual definition of the run-time environment by the generated scanner for different target languages.

D.1 'c'

Types

```
typedef enum smScScanner {
     sm_vocabularyname_scannername_Scanner = scannercode,
          ...
} smScScanner;

typedef enum sm_scannerName_ScToken {
     sm_vocabularyname_tokenname_Scanner = tokencode,
          ...
} sm_scannerName_ScToken;

typedef struct smScContextItem *smScContext;
typedef struct smScContextItem {
     int smSize;
     int smLength;
     unsigned char *smLength;
     int smLine;
     int smColumn;
     int smPosition;
     int smNextLine;
     int smNextColumn;
     int smNextPosition;
     smScScanner smScanner;
     code from context section
} *smScContext, smScContextItem;
```

Functions

```
int smScan(smThis,smToken)
     smScContext smThis;
     %%(tokenType) *smToken;

smScContext smScNew(smScanner)
     smScScanner smScanner;

void smScDelete(smThis)
     smScContext smThis;

int smScSkip(smThis,smLength)
     smScContext smThis;
     int smLength;

int smScModify(smThis,smBuffer,smLength)
     smScContext smthis;
     int unsigned char *smBuffer;
     int smLength;

int smScCopy(smThis,smBuffer,smFrom,smTo)
     smScContext smThis;
     unsigned char *smBuffer;
```

```
        int smFrom;
        int smTo;

    int smScMapCopy(smThis,smBuffer,smFrom,smTo)
        smScContext smThis;
        unsigned char *smBuffer;
        int smFrom;
        int smTo;

    void smScScrScanner(smScanner smOnOff)
        smScScanner smScanner;
        int smOnOff;

    void smScScrToken(smScanner,smToken,smOnOff)
        smScScanner smScanner;
        sm_vocabularyname_Token smToken;
        int smOnOff;

    void smScScrRule(smScanner,smToken,smOnOff)
        smScScanner smScanner;
        sm_vocabularyname_Token smToken;
        int smOnOff;
```

D.2 'ansi-c' and 'c++'

This interface is used both for 'ansi-c' and 'c++'.

Types

```
typedef enum smScScanner {
        sm_vocabularyname_scannername_Scanner = scannercode,
            ...
} smScScanner;

typedef enum sm_scannerName_ScToken {
        sm_vocabularyname_tokenname_Scanner = tokencode,
            ...
} sm_scannerName_ScToken;

typedef struct smScContextItem *smScContext;
typedef struct smScContextItem {
        int smSize;
        int smLength;
        unsigned char *smLength;
        int smLine;
        int smColumn;
        int smPosition;
        int smNextLine;
        int smNextColumn;
        int smNextPosition;
        smScScanner smScanner;
        code from context section
} *smScContext,smScContextItem;
```

Functions

```
int smScan(
        smScContext smThis,
        %%(tokenType) *smToken)
```

```
smScContext smScNew(
     smScScanner smScanner)

void smScDelete(
     smScContext smThis)

int smScSkip(
     smScContext smThis,
     int smLength)

int smScModify(
     smScContext smThis,
     int unsigned char *smBuffer,
     int smLength)

int smScCopy(
     smScContext smThis,
     unsigned char *smBuffer,
     int smFrom,
     int smTo)

int smScMapCopy(
     smScContext smThis,
     unsigned char *smBuffer,
     int smFrom,
     int smTo)

void smScScrScanner(
     smScScanner smScanner,
     int smOnOff)

void smScScrToken(
     smScScanner smScanner,
     sm_vocabularyname_Token smToken,
     int smOnOff)

void smScScrRule(
     smScScanner smScanner,
     sm_vocabularyname_Token smToken,
     int smOnOff)
```

D.3 Object oriented 'c++' (proposal)

This is a suggestion how an object-oriented 'c++' interface may look like. Currently there exist no such interface. 'c++' users should use the 'ansi-c' interface to generate source code compilable with 'c++' compilers.

Types

```
typedef enum smScScanner {
     sm_vocabularyname_scannername_Scanner = scannercode,
          ...
} smScScanner;

typedef enum sm_scannerName_ScToken {
     sm_vocabularyname_tokenname_Scanner = tokencode,
          ...
} sm_scannerName_ScToken;
```

```
class smScContextItem {
public:
     int smSize;
     int smLength;
     unsigned char *smLength;
     int smLine;
     int smColumn;
     int smPosition;
     int smNextLine;
     int smNextColumn;
     int smNextPosition;
     smScScanner smScanner;
public:
     int smScan(
          %%(tokenType) *smToken);
     smScContext(
          smScScanner smScanner);
     ~smScContext();
     int smScSkip(
          int smLength);
     int smScModify(
          unsigned char *smBuffer,
          int smLength);
     int smScCopy(
          unsigned char *smBuffer,
          int smFrom,
          int smTo);
     int smScMapCopy(
          unsigned char *smBuffer,
          int smFrom,
          int smTo);
     declarations from context section
};

void smScScrScanner(
     smScScanner smScanner,
     int smOnOff)

void smScScrToken(
     smScScanner smScanner,
     sm_vocabularyname_Token smToken,
     int smOnOff)

void smScScrRule(
     smScScanner smScanner,
     sm_vocabularyname_Token smToken,
     int smOnOff)
```

Part IV

ListerMaker Reference Manual

1 INTRODUCTION 197

2 CONCEPTS AND ASSUMPTIONS 198

2.1 ListerMaker and Lister 198
2.2 Source Positions 198
2.3 Messages 198
2.4 Error Message Templates and Insert Strings 199
2.5 Severities 199
2.6 Listing Types 200

3 LISTER PRODUCTION 201

3.1 The ListerMaker Description File 201
3.1.1 Lexical items 201
3.1.2 The Options Section 202
3.1.3 The Messages Sections 204
3.2 The ToolMaker Common Description File 204

4 THE LISTERMAKER COMMAND 206

4.1 Parameters 206
4.2 Options 206

5 LISTER RUN-TIME USAGE 208

5.1 Principles of Operation 208
5.1.1 Phases 209
5.1.2 Include-handling 210
5.1.3 Multiple Input Files 210
5.1.4 Separate Listing of Multiple Input Files 211
5.2 Run Time Interface 211
5.2.1 Constant: lmSEPARATOR 212
5.2.2 Type: lmMessages 212
5.2.3 Type: lmSev 212
5.2.4 Constant: sevALL 212
5.2.5 Type: lmTyp 212
5.2.6 Constant: liTINY 213
5.2.7 Constant: liFULL 213
5.2.8 Function: lmLiEnter() 213
5.2.9 Function: lmLiExit() 213
5.2.10 Function: lmLiInit() 213
5.2.11 Function: lmList() 214
5.2.12 Function: lmListm() 214
5.2.13 Function: lmListsi() 215
5.2.14 Function: lmLists() 215
5.2.15 Function: lmListse() 215
5.2.16 Function: lmLog() 215

5.2.17 Function: lmLogv() 216
5.2.18 Function: lmMsg() 216
5.2.19 Function: lmLiOff() 216
5.2.20 Function: lmLiOn() 217
5.2.21 Function: lmLiPage() 217
5.2.22 Function: lmLiPrint() 217
5.2.23 Function: lmLocSeverity() 217
5.2.24 Function: lmResLocSeverity() 217
5.2.25 Function: lmSeverity() 218
5.2.26 Function: lmSkipLines() 218
5.2.27 Function: lmLiTerminate() 218

5.3 Messages Templates 218
5.3.1 Insertion Markers 218
5.3.2 Reserved Numbers 219

A Appendix: THE PL/0 EXAMPLE 220

A.1 pl0.lmk - The ListerMaker Description File 220

B Appendix: ERROR MESSAGES 221

B.1 Message Explanations 221

C Appendix: TARGET LANGUAGE DETAILS ..222

C.1 'c' 222
C.2 'ansi-c' and 'c++' 224

1 INTRODUCTION

The ListerMaker is a component in ToolMaker, SoftLabs tool kit for software tool construction. ListerMaker is the error message handler and listing writing tool in that kit. ListerMaker gives a software tool writer facilities to

- collect all messages produced both during lexical, syntactical and semantic analysis of the input
- produce listings of the input and the messages in various formats
- run-time selectable message template sets, allowing e.g. multiple language support
- tuned to work with the other components in ToolMaker, giving easy implementation of a complete front-end of a compiler or other analysis tool.

Familiarity with compiler construction, and the principles of operation of the other tools in ToolMaker is somewhat assumed, although not strictly necessary.

2 CONCEPTS AND ASSUMPTIONS

2.1 ListerMaker and Lister

ListerMaker is a software tool to produce *Listers*. The term Lister is used to describe a software component or subroutine package designed to handle messages issued during analysis of an input, typically error and warning messages. A Lister also allows production of listing files containing the input and any associated messages. It is also possible to retrieve the messages one by one from the Lister.

ListerMaker is the tool and Lister is thus the subroutine package produced by that tool.

2.2 Source Positions

The most important concept in ListerMaker is the *source position*. A source position is an identification of a position in the source (input) worked upon by the application (i.e the program written by you, the tool writer). Although perfectly usable anyway, conceptually this is the first assumption made by ListerMaker, that the application is reading its input from a file. A source position is a structure containing a column, a line and optionally a file number.

A source position is sent to Lister together with every message logged. This makes it possible for the Lister to find and show the source corresponding to the error and to place the message at the correct point in the output. A message that should be placed at the end of the listing of a file should have the line component set to 0 (zero). All such messages are collected and printed after listing the file. In case multiple files are handled, global messages, i.e. messages pertaining to the complete input can be collected and shown after all files are listed if the file component is set to -1.

> *Note:* The automatic source position calculation of ScannerMaker can be used to create the line and column part of the source position information.

2.3 Messages

A *message* is actually a number of different things. The first meaning of a message is *one number and corresponding text in the Message section in the description file* (see below). This type of message is a template for actual messages output to the end-user.

The second use of the term is *the call to Lister to log a message*, this call carries among other things a message number referring to the message in the previous sense, i.e it refers to a template found in the description or message file.

The third use is *the actual presented text* together with the source where the error was discovered. This is produced by the Lister using a call to a list function after all messages have been logged and shown on the terminal or the screen.

A final observation on error messages is that although the term is error message it does not actually have to indicate errors (other classes of messages handled by a Lister is Informational, Warning, Fatal and System messages) but is used for convenience throughout this document. ListerMaker does not place any restrictions or assumptions on what to do when a message is logged, the recovery, abortion of execution or other action is totally up to the implementor.

2.4 Error Message Templates and Insert Strings

As described above an error message starts out as a template in the description file. These templates consist of a number used to identify the message and a string which is the actual message each taking up one line in the error message file. For example

```
102   '%1 deleted.';
```

The message string may contain insertion markers ('%d', where d is a digit). This allows a message to be customised during run-time by inserting additional information in the message clarifying the meaning, such as the name of an identifier. The needed number of insert strings must be concatenated and sent to Lister in the log call. Lister will then extract the insert strings and insert them at the appropriate places in the message before presenting it to the user.

2.5 Severities

A logged message is also associated with a severity code. This code is not implied by the message code, as it sometimes might be handy to be able to log a message with the same code but with different severities. An example of this is the insertion of a symbol by a parser, if the symbol was just a comma or semicolon, this might be considered a Warning, but if the inserted symbol was an identifier it surely must be an Error as the parser has no idea as to what identifier the programmer meant.

ListerMaker (and the Lister) defines the following severity levels (in order of increasing severity):

```
sevINF, sevWAR, sevERR, sevFAT, sevSYS
```

corresponding to Informational, Warning, Error, Fatal and System messages. Any semantic meaning of these, except for their ordering, is not assumed by the Lister, it is only used as a classification of the messages.

The most serious severity logged so far may be read by the function ***lm***`Severity()`.

2.6 Listing Types

In order to direct Lister to produce the selected kinds of information in the listing the generated Lister also defines an enumerated type. One of the parameters to ***lm***`List()` is a set of values from this type.

Value	Description
`liSUM`	Print a summary of the number of messages found
`liERR`	Print source lines with messages associated to them in the list
`liOK`	Print source lines without any messages in the list
`liINCL`	List and show (as appropriate) included files
`liHEAD`	Print a header on top of all pages

Lister also has a shorthand notation for two common combinations of these, `liTINY`, which is a tiny listing including a header and source for just those lines that have messages associated to them and a summary, and `liFULL`, which also shows the source lines without messages. Normally a tiny list is useful to present on the screen, and a full list if listing to a file.

3 LISTER PRODUCTION

3.1 The ListerMaker Description File

The main information necessary to produce a Lister is the message templates. These are normally written in a description file used by ListerMaker.

To inform ListerMaker how your Lister is to work there is a number of options that you may choose. These include for example whether to include routines for list production and the prefix for all the externally visible functions in the produced Lister.

After setting up a directory using ***toolmake*** there should exist a default description file prepared with all necessary options set to their default values and some example messages. An example of a ListerMaker description file is included in appendix A, *THE PL/0 EXAMPLE*, on page 220.

The structure of the ListerMaker description file is

```
<description file> ::=
     <toolmaker sections>
     <message section> { <message section> }

<toolmaker sections> ::=
     [ <options section> ]
     { <import section> | <srcp section> }
```

The *import* and *srcp sections* are further described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. These sections are normally located in the ToolMaker description file and should be put in the ParserMaker description file only when developing a stand-alone Lister.

3.1.1 Lexical items

Symbols in the ListerMaker description language are constructed from using upper case letters from the ISO-8859-1 character set, lower case letters (ISO-8859-1) and digits.

```
<letter> ::= <upper case letter> | <lower case letter>

<digit> ::=
       '0' | '1' | '2' | '3' | '4'
     | '5' | '6' | '7' | '8' | '9'

<number> ::= <digit>+

<name> ::= <letter> {<letter> | <digit> | '_'}

<quoted string> ::=
     ''' {<letter> | <digit> | <special character>} '''
```

3.1.2 The Options Section

```
<options section> ::=
     '%%OPTIONS' <directive> {<directive>} ['%%END']

<directive> ::=
       <common directive>
     | <listings directive>
     | <include directive>
     | <message directive>
     | <limit directive>
```

In the options section various options can be set. The available options for ListerMaker are described below. The term default describes the initial setting unless otherwise stated in the ToolMaker Common Description file when appropriate (see *The ToolMaker Description File* in the *ToolMaker System Description*, page 27).

These options can be overridden by options given on the command line when ListerMaker is invoked (see *THE LISTERMAKER COMMAND* on page 206). If an option is not specified its default value is used. The default options are:

```
No Verbose;
Target 'ansi-c';
Os 'SunOS';
Prefix 'lm';
Library '$TMHOME/lib/ansi-c';
Escape '\';
Width 78;
Height 60;
Generate source;
No Force;
Listings single;
Include;
Limit 100;
Messages embedded;
```

Note however that the settings of common options in the ToolMaker Common Description file influences the default values (see *The Options Section* in the *ToolMaker System Description*, page 30, and *Prefix Management* in the *ToolMaker System Description*, page 42).

3.1.2.1 Common Directives

```
<common directive> ::=
       <prefix directive>
     | <escape directive>
     | <target directive>
     | <os directive>
     | <library directive>
```

The common directives are directives available for all Makers in the ToolMaker kit. For a detailed description of these refer to *The Options Section* in the *ToolMaker System Description*, page 30. The directives listed above are the directives relevant for ListerMaker, and if used overrides settings and default values from the ToolMaker Common Description file.

The prefix directive does not inherit its default value, instead it defaults to '**lm**' if not explicitly set in the **.tmk** file. If set in the ToolMaker Common Description file and *not* used in the ListerMaker Description file it defaults to the system prefix (the value set in the ToolMaker Common Description file).

3.1.2.2 The Listings Directive

```
<listings directive> ::=
       'LISTINGS' <listings option>
              {',' <listings option>} ';'
     | ['NO'] 'LISTINGS' ';'

<listings option> ::=
     'SINGLE' | 'MULTIPLE' | 'SEPARATE'
```

The listings directive tells ListerMaker to include the functions for producing listings in the generated Lister module. In cases where only message by message retrieval using the **lm**`Msg()` function is used the `NO LISTINGS` option may be specified to reduce the amount of code in the Lister.

Under certain circumstances it is necessary to make a list of multiple source files, for example if the application is checking the interfaces between program modules. If this directive is specified, ListerMaker will generate a **lm**`Listm()` function, which may be used instead of **lm**`List()` to achieve this. See *Multiple Input Files* on page 210.

The functions to use for separate listings of multiple input files (see *Separate Listing of Multiple Input Files* on page 211) is only generated into the Lister if the value `SEPARATE` is selected.

The default value is.

```
Listings SINGLE;
```

3.1.2.3 The Include Directive

```
<include directive> ::=
     ['NO'] 'INCLUDE' ';'
```

The include directive specifies if using and nesting source files shall be allowed. If `NO INCLUDE` is specified, file handling will not be included in the generated Lister.

The default value is:

```
Include;
```

3.1.2.4 The Limit Directive

```
<limit directive> ::=
     'LIMIT' <number> ';'
```

The `Messagelimit` directive indicates the maximum number of messages possible to log using ***lm***`Log()`. Any integer greater than 0 (zero) is allowed. Note that also the number of calls to ***lm***`LiEnter()`, ***lm***`LiExit()`, ***lm***`LiOff()`, ***lm***`LiOn()`, ***lm***`LiPage()`, ***lm***`SkipLines()` (i.e. all COLLECTING phase functions) are included in this calculation.

The default value is:

```
Limit 100;
```

3.1.2.5 The Message Directive

```
<message directive> ::=
     'MESSAGE' <target option> ';'

<message option> ::=
     'FILE' | 'EMBEDDED'
```

The message directive tells ListerMaker whether the messages shall be put in a separate file to be read at run-time or embedded in the generated Lister source code.

The default value is:

```
Message EMBEDDED;
```

3.1.3 The Messages Sections

```
<messages section> ::=
     '%%MESSAGES' <name>
     [<message> {<message>}]
     ['%%END']

<message> ::= <number> <quoted string> ';'
```

Multiple message sections may be specified. Using the generated initialisation call (***lm***`LiInit()`) the section used may be selected during run-time. Example:

```
%%MESSAGES english
     10 'Syntax error.';
     11 'Illegal symbol.';
%%MESSAGES swedish
     10 'Syntaxfel.';
     11 'Felaktig symbol.';
%%END
```

3.2 The ToolMaker Common Description File

Unless ListerMaker is the only Maker used, common declarations of the source position and the token structures should be placed in the ToolMaker

Common Description file which is described in *THE TOOLMAKER DESCRIPTION FILE* in the *ToolMaker System Description*, page 30. Otherwise these two sections may be specified in the ListerMaker Description file, removing any need for the ToolMaker Common Description file.

4 THE LISTERMAKER COMMAND

The invocation of ListerMaker is performed by a command of the form

```
lmk [-help] [<options> ...] <file>
```

The special option `-help` gives a short help of available options and their meaning.

4.1 Parameters

The ListerMaker command takes only one argument, the name of a description file. The description file contains a description of the lister to be generated using definitions as described in *The ListerMaker Description File* on page 201. The argument may appear anywhere on the command line except between an option name and its arguments. The default extension for the ListerMaker description file is **.lmk**.

4.2 Options

One or more options may be specified on the command line overriding any specified options in the *options section* in the description file.

All options are available as command line options, the general format is:

```
-[-]optionname [optionvalue]
```

The option names corresponds exactly to the names given in the section *The Options Section* on page 202. The option value (if required) corresponds to allowed values of the option. To turn an option on, one dash is used, to turn an option off two dashes are used. For example, to turn the verbose option on, use `-verbose`, to turn it off, use `--verbose`, which is equivalent to specifying `NO VERBOSE` in the options section.

Please refer to the corresponding options and directives in *The Options Section* on page 202 for a detailed description of the various options. The general command line option format is described in *Command Line Option Format* in the *ToolMaker System Description*, page 40.

The special option

```
-help
```

produce a verbose listing of the usage format of the command. Each argument and option are given a short explanation. Refer to *The Options Section* on page 202 for details on the various options.

In addition the following special option is available from the command line:

```
-lmt <file>
```

This options may be used to direct ListerMaker to write the intermediate tables on a file other than the default.

5 LISTER RUN-TIME USAGE

This chapter describe the functions in the run time part of ListerMaker, the Lister, and how your application accesses these functions.

5.1 Principles of Operation

The first thing to do is to initiate the listing system. This is done by calling the function `lmLiInit()`, preferably at the beginning of the main program. This call includes the name of the input file and the string to insert in the header of any listing files produced later and the message section that is to be used for templates. If message templates were placed in a separate file (by the use of the appropriate option) the name of the file containing the error message templates should also be given in this call.

The next step is to start analysis of the input, for example by initiating a ScannerMaker generated scanner and calling the parsing routine in the ParserMaker generated parser. During this analysis lexical and syntactical errors discovered are logged by the error handling procedures in the error handler module used by ParserMaker (see *ParserMaker Reference Manual*). The supplied ListerMaker description file contains error message templates for the errors automatically handled here.

Messages are logged using the function `lmLog()`. This function is also used by the standard error handler module in the parser. Of course this module may be replaced by code written by you, but as it stands it is tuned to directly work with a ListerMaker produced Lister.

If you are not using a ParserMaker generated parser you might still want to use Lister, and this is of course possible. Just use `lmLog()` to log the messages. This is also what you do for example during later passes of your application when you want to log a message.

The last step in using Lister is to use it to present the messages to the user. This is performed by calling `lmList()` which creates a listing presented in a file or on the screen. The information contained in the listing is controlled by parameters to `lmList()`, examples of information possible to select are whether only messages or source lines corresponding to the errors or all lines should be shown. This is performed by re-reading the input files and for each line produce the appropriate output (input lines and/or messages).

An alternate way of presenting the messages for a user is to use the `lmMsg()` function. This function will return the message text for a any selected message together with its source position. This information may then be used to position the message text on a screen for example.

If multiple files are to be listed the source position structure *must* contain a file component. This field should be initialised with a unique value for each in-

stance of input files. To use the value from a simple counter which is incremented for each newly opened file is normally adequate.

5.1.1 Phases

Lister works in three different phases, UNINITIALISED, COLLECTING and RETRIEVING. During the UNINITIALISED phase no calls except to initiate the Lister is legal. Calls to procedures logging messages is allowed in the COLLECTING phase, and calls for producing lists, retrieving messages and printing text is only allowed in the RETRIEVING phase. Transition between the COLLECTING and RETRIEVING phases is performed by the first call to `lmList()`, `lmListm()`, `lmListsi()` or `lmMsg()`.

To sum up, the following functions are available in each phase:

INITIATING:

- *lm*LiInit()

COLLECTING:

- *lm*Log()
- *lm*Logv()
- *lm*LiOn()
- *lm*LiOff()
- *lm*LiPage()
- *lm*Severity()

RETRIEVING:

- *lm*List()
- *lm*Msg()
- *lm*SkipLines()
- *lm*LiPrint()
- *lm*Severity()
- *lm*LiTerminate()

5.1.2 Include-handling

In some cases a method of handling nested files is needed. This is for example the case with so called include-files in some programming languages. ListerMaker supplies functions to handle this feature.

Suppose that we want to create a 'c'-preprocessor. The preprocessor has a directive, `#include`, which allows inclusion of secondary files containing 'c' source.

To solve the problem of creating lists of such input using ListerMaker the function ***lm***`LiEnter()` is used. When encountering the `#include` directive this function should be called with the name of the new (included) file and a source position where the file should be inserted (normally at the first position on the line following the `#include` directive.

During the RETRIEVING phase any messages logged using source positions indicating this included file will be shown (or returned using ***lm***`Msg()`) before messages occurring after the `#include` directive in the top level file. So all messages are sorted correctly.

Note that the `file` component of the source position structure should differentiate between every instance of an included file since even if the same file is included more than once the context is different each time and different messages may be generated for every instance depending on the context.

The following extra functions are available if the `include` option is used:

COLLECTING:

- *lm*LiEnter()
- *lm*LiExit()

5.1.3 Multiple Input Files

Another major feature of Lister is the ability to handle listing of multiple top-level input files. These files have no hierarchical relations as is the case with included files, as described above.

As opposed to the include file handling this is used when multiple files are worked upon at the same time, for example a file compare program or a module interface verifier. This kind of applications require that a small number of related input files are analysed (the files having no hierarchical relation to each other). When creating a listing the files should appear in the same listing file, but separated, not nested.

This requires an array of file names, each index corresponding to the file number used in the source position structure, to be passed to Lister, as is the case in the parameter list for the *lm*`Listm()` function, which is designed for this purpose. In this case the file numbers for the top level files must be from zero and up. Any further files may have any number.

> *Note:* this feature is possible to use together with the include-feature, meaning that a number of top level files are listed together, each having (possibly) nested input files.

The following extra functions are available if the `Listings multiple;` option is used:

RETRIEVING:

- *lm*Listm()

5.1.4 Separate Listing of Multiple Input Files

The listing of multiple input files can also be handled separately as opposed to the multiple handling described above. This means that a call to a function (*lm*`Lists()`) must be performed for each of the files. This makes it possible to handle output (using *lm*`LiPrint()`) between listings of each file.

The following extra functions are available if the `Listings separate;` option is used:

COLLECTING:

- *lm*LocSeverity()
- *lm*ResLocSeverity()

RETRIEVING:

- *lm*Listsi()
- *lm*Lists()
- *lm*Listse()

5.2 Run Time Interface

Below all functions in a Lister are described with their default prefix, *lm*. By using the option `Prefix` this may be changed to any selected string.

5.2.1 Constant: lmSEPARATOR

This character constant contains the value to be used as separator between concatenated insertion strings. Thus the operation to send two insertion strings to a message in one lmLog() call is to concatenate the two strings separated with the value lmSEPARATOR, and then to send the resulting string as the insertion string in the lmLog() call.

5.2.2 Type: lmMessages

This type contains the enumerated values of the names of the sections given in the Message sections in the description file. There is one value for each named Message section constructed by prepending the lister prefix and an underscore, and appending an underscore and the text 'Messages' to the name given to the section, e.g. if the section was called 'english' the corresponding constant is called

```
lm_ENGLISH_Messages
```

One of these values must be transferred in the call to the lmLiInit() function to indicate which set of message templates to use.

5.2.3 Type: lmSev

This enumerated type has all severities exported from Lister as its value set. Values from this type may be combined to form sets used in lmList().

Value	Description
sevOK	OK severity, no message.
sevINF	Informational message.
sevWAR	Warning message.
sevERR	Error message.
sevFAT	Fatal error message.
sevSYS	System error message.

5.2.4 Constant: sevALL

An lmSev constant containing the set of all severities (sevOK, sevINF, sevWAR, sevERR, sevFAT, sevSYS).

5.2.5 Type: lmTyp

This enumerated type is used to indicate selected information to include in the listing file (or on the screen).

Value	Description
liSUM	List a summary of found messages.
liERR	List source for lines with messages.

`liOK` List source for lines without messages.
`liINCL` List source lines and messages from included files as indicated by the presence of `liERR` and `liOK` flags.
`liHEAD` Print a header on each page in the listing file.

5.2.6 Constant: liTINY

A set of ***lm***`Typ` values appropriate for a tiny list, for example on the screen. Contains `liSUM`, `liERR`, `liINCL` and `liHEAD`.

5.2.7 Constant: liFULL

A set of ***lm***`Typ` values appropriate for listing to a list file. Also contains `liOK`, so as to also show source lines for which there are no messages.

5.2.8 Function: ***lm***LiEnter()

```
lmLiEnter(srcp, start, file)

srcp : IN %%(srcpType)
start : IN %%(srcpType)
file : IN STRING
```

If so configured the generated Lister is capable of handling nested files (include files in programmer terminology). A call to the ***lm***`LiEnter()` function during message collection indicates where the included file should be inserted. The `srcp` parameter indicates the source position where the entered file should be inserted, and `start` where in the entered file to begin, normally this should indicate the first column on the first line. The file component should be set to a unique value indicating this instance of the included file. ***lm***`LiEnter()` is only callable in COLLECTING phase.

5.2.9 Function: ***lm***LiExit()

```
lmLiExit(srcp)

srcp : IN %%(srcpType)
```

Prematurely exit from a nested file. Using this function is analogous to saying that the end of the file is earlier than the physical end of file. `srcp` is the source position where to place the simulated end of file. Should only be used in special cases and is only allowed in COLLECTING phase.

5.2.10 Function: ***lm***LiInit()

```
lmInit(header, source, section, message)

header : IN STRING
source : IN STRING
section : IN lmMessage
message : IN STRING
```

Initiates the listing system. Only allowed in UNINITIALISED phase (the call changes phase to COLLECTING). `header` is the string to insert in the page header on every page of any listing file. This could for example contain a version identification. The file name of the top level source file is passed in the `source` parameter, the constant indicating which section of messages to use in `section` and the name of the message template file in `message`.

> *Note:* If message templates are placed in the source instead of in a separate message file (by use of the `'Messages EMBEDDED'` option) the last parameter, `message`, is not available and should not be given.

5.2.11 Function: *lm*List()

```
lmList(outfnm, lines, columns, listtype, severities)

outfnm : IN STRING
lines   : IN INTEGER
columns : IN INTEGER
listtype : IN SET OF lmTyp
severities : IN SET OF lmSev
```

The *lm*`List()` function produces a list consisting of the requested information, such as source lines, error messages and page headers on a specified file or on the terminal. It is callable only in the COLLECTING (the first call changes phase to RETRIEVING) and RETRIEVING phases. The file name for the list is passed in `outfnm`, if it is equal to "" (the empty string) the result is a listing on the screen (standard error). `lines` specifies the numbers of lines per page on the output file, if equal to 0 (zero) no paging is performed. `columns` specifies the number of output columns. A set of values from the *lm*`Typ` values is sent in the `listtype` parameter specifying the information to be listed. `severities` should contain a set of values from the *lm*`Sev` type indicating which severities to include in the listing.

5.2.12 Function: *lm*Listm()

```
lmListm(outfnm, lines, columns, listtype, severities,
        files)

outfnm : IN STRING
lines : IN INTEGER
columns : IN INTEGER
listtype: IN SET OF lmTyp
severities : IN SET OF lmSev
files : IN ARRAY OF STRING
```

Same as *lm*`List()` but allows for listing of multiple input files. This allows an application to e.g. analyse a number of input files and still produce one listing file. Note that this is not the same as the include handling (see *Include-handling* on page 210 and *Multiple Input Files* on page 210 for a comparison). Instead each input file is listed separately with a page break between each one. The extra parameter `files` is an array of strings with the file names of the files to list. The last element must be a NULL pointer.

This function is only available if the `Listings multiple;` is used.

5.2.13 Function: *lm*Listsi()

```
lmListsi(outfnm, lines, columns, listtype, severities)

outfnm : IN STRING
lines : IN INTEGER
columns : IN INTEGER
listtype: IN SET OF lmTyp
severities : IN SET OF lmSev
```

This function initialises the lister to start separate listing (see *Separate Listing of Multiple Input Files* on page 211). This function does not produce a complete listing, instead the function `lmLists()` needs to be called once for each file, and the terminating function `lmListse()` after completing the listing.

The parameters have the same interpretation as for `lmList()`.

5.2.14 Function: *lm*Lists()

```
lmListse(severities, fno, fnm)

severities : IN SET OF lmSev
fno : INTEGER
fnm : STRING
```

The function performs listing of *one* file while using separate listing (see *Separate Listing of Multiple Input Files* on page 211). A call to `lmListsi()` must first be performed to set it up, then multiple calls to `lmLists()` can be performed until all files have been handled. Between calls other retrieving phase functions can be called (e.g. `lmLiPrint()` to print extra information).

5.2.15 Function: *lm*Listse()

```
lmListse(severities)

severities : IN SET OF lmSev
```

This function terminates the listing of separate files (see *Separate Listing of Multiple Input Files* on page 211) by printing the global messages. The severity parameter indicates which severities to include in the global messages section.

5.2.16 Function: *lm*Log()

```
lmLog(srcp, ecode, sev, istrs)

srcp : IN %%(srcpType)
code : IN INTEGER
```

```
sev : IN lmSev
istrs : IN STRING
```

Collects (logs) a message together with source position information (`srcp`) to be expanded and sorted for later retrieval. It is only callable in the COLLECTING phase. `code` identifies the message template from the message template file to be used. The severity associated with the message is indicated in `sev`. Concatenated insert strings are passed in `istrs`, i.e. one string for each insertion marker in the template, separated by the `lm`SEPARATOR character constant.

5.2.17 Function: lmLogv()

```
lmLogv(srcp, ecode, sev, ...)

srcp : IN %%(srcpType)
code : IN INTEGER
sev : IN lmSev
```

Corresponds to the function `lm`Log() but uses the 'c'-language feature of variable number of arguments. This greatly simplifies the logging of messages having multiple insert strings.

Note: The function is only available in the 'c' target languages.

Note: The parameter list must be terminated by a NULL value.

5.2.18 Function: lmMsg()

```
lmMsg(i, srcp, msg) : INTEGER

i : IN INTEGER
srcp : OUT %%(srcpType)
msg : OUT STRING
returns INTEGER
```

Retrieves logged messages one by one. For each call the `i`'th message is retrieved, the source position of the message is returned in the out parameter `srcp` and the message text in `msg`. `lm`Msg() returns the index, `i`, if the message was found, else a 0 (zero) is returned. This can be used to determine that there where no more messages to retrieve. Allowed in the RETRIEVING and COLLECTING phases (first call changes phase to RETRIEVING).

5.2.19 Function: lmLiOff()

```
lmLiOff()Off(srcp)

srcp : IN %%(srcpType)
```

The function of `lm`LiOff() is to turn off the listing from a specified source position, `srcp`. Only allowed in COLLECTING phase.

5.2.20 Function: lmLiOn()

```
lmLiOn(srcp)

srcp : IN %%(srcpType)
```

Turns listing on again from source position `srcp`, after it has been turned off. Callable only in COLLECTING phase.

5.2.21 Function: lmLiPage()

```
lmLiPage(srcp, lines)

srcp : IN %%(srcpType)
lines : IN INTEGER
```

Produces a conditional or unconditional page break in the output listing at a specified source position, `srcp`. If there are less than `lines` lines more available on the current listing page a page break is performed otherwise not. Zero means an unconditional break. `lmLiPage()` should only be used in COLLECTING phase.

5.2.22 Function: lmLiPrint()

```
lmLiPrint(line)

line : IN STRING
```

Print a line in the current output file. After each call to `lmList()` (or between consecutive calls to `lmListse()`) it is possible to print additional lines in the output file. As long as `lmList()` is not called again (or `lmLiTerminate()`!) each line will be appended to the listing file. Each call prints the contents of the parameter and performs a new line in the listing file or on the terminal. Page and line breaks are handled correctly. Allowed in RETRIEVING phase only. `line` is the string to be printed.

5.2.23 Function: lmLocSeverity()

```
sev = lmLocSeverity()

returns lmSev
```

The function `lmLocSeverity()` may be used to read the highest severity logged since the reset of a special local severity variable (see *Function: lmResLocSeverity()* on page 217) This function is callable both in COLLECTING and RETRIEVING phases and only available if `Listings separate;` is used.

5.2.24 Function: lmResLocSeverity()

```
lmResLocSeverity()
```

The **lm**ResLocSeverity() resets the special local severity variable available when the Listings separate; option is in effect.

> *Note:* the implementor is completely responsible for the use and resetting of the local severity variable.

5.2.25 Function: **lm**Severity()

```
sev = lmSeverity()

returns lmSev
```

To read the highest severity logged so far the procedure **lm**Severity() may be used which returns a value from the **lm**Sev value set. This function is callable both in COLLECTING and RETRIEVING phases.

5.2.26 Function: **lm**SkipLines()

```
lmSkipLines(lines)

lines : IN INTEGER
```

Performs a conditional page break in the output file. As with **lm**LiPrint() this function performs its output in the current output file or on the terminal. **lm**SkipLines() corresponds to the COLLECTING phase function **lm**LiPage(), but may only be used in RETRIEVING phase. lines contains the number of lines at least available to *not* make a page break.

5.2.27 Function: **lm**LiTerminate()

```
lmLiTerminate()
```

Terminates the Lister. No more lists may be produced. Lister is prepared for a new **lm**LiInit() call. Allowed only in the RETRIEVING phase (the call changes phase to UNINITIALISED).

5.3 Messages Templates

5.3.1 Insertion Markers

An insert string supplied in a **lm**Log() call is inserted into the message template fetched from the message file. The insert string may contain multiple strings separated by the special character **lm**SEPARATOR exported by Lister. The insertion marker is the character '%' followed by a number indicating which part of the string and should be placed in the template where the corresponding part of the insert string should be inserted. Thus the insert string supplied to **lm**Log() must contain at least as many **lm**SEPARATOR separated parts as there are insertion markers in the message text.

5.3.2 Reserved Numbers

The following message numbers are reserved and should always be available if the generated Lister is used to produce listings (i.e. it is not necessary if only message retrieval using `lmMsg()` is used).

1) Heading string (may include insertion markers, position of insertion marker indicates where to insert any insertion strings sent in the `lmLiInit()` call).

2) Message text for no errors or warnings (`sevERR` or `sevWAR`).

3) Message text for no detected warnings (`sevWAR`).

4) Message text for no detected errors (`sevERR`).

5) Message text for number of detected informational messages (`sevINF`). An insertion marker is used to indicate where to insert the number.

6) Message text for number of detected warning messages (`sevWAR`). An insertion marker is used to indicate where to insert the number.

7) Message text for number of detected error messages (`sevERR`). An insertion marker is used to indicate where to insert the number.

8) Message text for maximum number of messages exceeded.

By altering the templates in the description file these system messages may be customized for different languages or other formats.

Note: The reserved message templates must be available in all message sections.

A THE PL/0 EXAMPLE

This example is a part of the overall example used throughout the ToolMaker documentation. In *ToolMaker System Description* there is a detailed walk-through of the main parts of the example. This appendix contains the files relevant for ListerMaker.

A.1 pl0.lmk - The ListerMaker Description File

```
------------------------------------------------------
-- pl0.lmk     Date: 1993-06-10/toolmake
--
-- pl0 - ListerMaker description file
--
------------------------------------------------------
-- Created:   1993-04-27/reibert@roo
-- Generated: 1993-06-10 12:47:14/toolmake v2,r0,c7
------------------------------------------------------

%%OPTIONS
    Include;

%%MESSAGES English
  0   'PL/0 Analysis Tool %1';
  1   'No warnings or errors detected.';
  2   'No warnings issued.';
  3   'No errors detected.';
  4   '%1 informational message(s) produced.';
  5   '%1 warning(s) issued.';
  6   '%1 error(s) detected.';
  7   'Maximum number of messages exceeded.';
100   'Parsing resumed.';
101   '%1 inserted.';
102   '%1 deleted.';
103   '%1 replaced by %2.';
104   'Syntax error, stack backed up.';
105   'Syntax error.';
106   'Parse stack overflow.';
107   'Parse table error.';
108   'Parsing terminated.';
199   'File %1 not found.';
%%END
```

B ERROR MESSAGES

For a description of the format of error messages refer to *Message Format* in the *ToolMaker System Description*, page 49.

B.1 Message Explanations

The following list gives a brief summary of the error messages and in some cases the actions that should be taken by the user. Messages with numbers less than 100 are messages common for all Makers. These are described in *Messages Explanations* in the *ToolMaker System Description*, page 50. Messages indicating license problems or limitations are described in *License Errors* in the *ToolMaker System Description*, page 51.

200 `Message section "%1" already defined.`

The indicated name was already used for another section of error messages.

300 `At least one messages section is required.`

At least one section with error messages must be defined.

C TARGET LANGUAGE DETAILS

This appendix lists the actual definition of run-time elements exported by ListerMaker. The following sections contain this information per target language.

`%%(srcpType)` denotes the type name defined for the source position structure in the *srcp section* in the ToolMaker control file (see appendix A, *THE PL/0 EXAMPLE*, on page 220 and *The ToolMaker Description File* in the *ToolMaker System Description*, page 27). Functions, types and constants prefixed by ***lm*** uses the `listerPrefix` (see *The Messages Sections* on page 204).

C.1 'c'

Types

```
typedef enum lmSev {
     sevOK  = (1<<0),
     sevINF = (1<<1),
     sevWAR = (1<<2),
     sevERR = (1<<3),
     sevFAT = (1<<4),
     sevSYS = (1<<5)
} lmSev;

typedef enum lmTyp {
     liSUM  = (1<<0),    /* Summary */
     liERR  = (1<<1),    /* Erroneous source lines */
     liOK   = (1<<2),    /* Correct source lines */
     liINCL = (1<<3),    /* Look also in PUSHed files */
     liHEAD = (1<<4)     /* Heading */
} lmTyp;

typedef enum lmMessages {
     lm_XXXX_Messages,
     ....
} lmMessages;
```

Constants

```
#define lmSEPARATOR ((char)0xff)

#define liTINY (liSUM|liERR|liHEAD|liINCL)
#define liFULL (liTINY|liOK)

#define sevALL (sevOK|sevINF|sevWAR|sevERR|sevFAT|sevSYS)
```

Functions

```
void lmLiInit(header, src, msect)
     char header[];
     char src[];
     lmMessages msect;
```

```
void lmLog(pos, ecode, sev, istrs)
     %%(srcpType) *pos;
     int ecode;
     lmSev sev;
     char istrs[];

void lmLiOff(pos)
     %%(srcpType) *pos;

void lmLiOn(pos)
     %%(srcpType) *pos;

void lmLiEnter(pos, start, fnm)
     %%(srcpType) *pos;
     %%(srcpType) *start;
     char fnm[];

void lmLiExit(%%(srcpType) *pos);

void lmLiPage(pos, lins)
     %%(srcpType) *pos;
     int lins;

lmSev lmSeverity();

lmSev lmLocSeverity();

void lmResLocSeverity();

void lmList(ofnm, lins, cols, typ, sevs)
     char ofnm[];
     int lins;
     int cols;
     lmTyp typ;
     lmSev sevs;

void lmListm(ofnm, lins, cols, typ, sevs, fnms)
     char ofnm[];
     int lins;
     int cols;
     lmTyp typ;
     lmSev sevs;
     char *fnms[];

void lmListsi(ofnm, lins, cols, typ)
     char ofnm[];
     int lins;
     int cols;
     lmTyp typ;

void lmLists(sevs, fno, fnm)
     lmSev sevs;
     int fno;
     char *fnm;

void lmListse(sevs)
     lmSev sevs;

int lmMsg(i, pos, msg)
     int i;
     %%(srcpType) *pos;
     char *msg);
```

```
void lmLiPrint(str)
     char str[];

void lmSkipLines(lins)
     int lins;

void lmLiTerminate();
```

C.2 'ansi-c' and 'c++'

Types

```
typedef enum lmSev {
     sevOK  = (1<<0),
     sevINF = (1<<1),
     sevWAR = (1<<2),
     sevERR = (1<<3),
     sevFAT = (1<<4),
     sevSYS = (1<<5)
} lmSev;

typedef enum lmTyp {
     liSUM  = (1<<0),    /* Summary */
     liERR  = (1<<1),    /* Erroneous source lines */
     liOK   = (1<<2),    /* Correct source lines */
     liINCL = (1<<3),    /* Look also in PUSHed files */
     liHEAD = (1<<4)     /* Heading */
} lmTyp;

typedef enum lmMessages {
     lm_XXXX_Messages,
     ....
} lmMessages;
```

Constants

```
#define lmSEPARATOR ((char)0xff)

#define liTINY (liSUM|liERR|liHEAD|liINCL)
#define liFULL (liTINY|liOK)

#define sevALL (sevOK|sevINF|sevWAR|sevERR|seFAT|sevSYS)
```

Functions

```
void lmLiInit(
     char header[],
     char src[],
     lmMessages msect);

void lmLog(
     %%(srcpType) *pos,
     int ecode,
     lmSev sev,
     char istrs[]);
```

```
void lmLogv(
     %%(srcpType) *pos,
     int ecode,
     lmSev sev,
     ...);

void lmLiOff(%%(srcpType) *pos);

void lmLiOn(%%(srcpType) *pos);

void lmLiEnter(
     %%(srcpType) *pos,
     %%(srcpType) *start,
     char fnm[]);

void lmLiExit(%%(srcpType) *pos);

void lmLiPage(
     %%(srcpType) *pos,
     int lins);

lmSev lmSeverity(void);

lmSev lmLocSeverity(void);

void lmResLocSeverity(void);

void lmList(
     char ofnm[],
     int lins,
     int cols,
     lmTyp typ,
     lmSev sevs);

void lmListm(
     char ofnm[],
     int lins,
     int cols,
     lmTyp typ,
     lmSev sevs,
     char *fnms[]);

void lmListsi(
     char ofnm[],
     int lins,
     int cols,
     lmTyp typ);

void lmLists(
     lmSev sevs,
     int fno,
     char *fnm);

void lmListse(lmSev sevs);

int lmMsg(
     int i,
     %%(srcpType) *pos,
     char *msg);

void lmLiPrint(char str[]);

void lmSkipLines(int lins);
```

```
void lmLiTerminate(void);
```

Part V

Toolmake
Reference Manual

1 INTRODUCTION 231

2 PRINCIPLES OF OPERATION 232

3 THE TOOLMAKE COMMAND 233

3.1 Parameters 233

3.2 Options 233

4 TOOLMAKE RUN-TIME USAGE 234

4.1 Subsystem Name 234

4.2 ToolMake Commands 234

4.2.1 GENERATE 234

4.2.2 QUIT 235

4.2.3 ALL 235

4.2.4 NONE 235

4.2.5 PARSER, SCANNER, LISTER, MAIN, MAKEFILE 235

4.2.6 LEVEL 235

4.2.7 LANGUAGE 235

4.2.8 VERBOSE 236

4.2.9 HELP 236

4.2.10 INFORMATION 236

A Appendix: ERROR MESSAGES 237

A.1 System Errors 237

A.2 Fatal Errors 237

A.3 Errors 237

A.4 Warnings 238

A.5 Informational Messages 239

B Appendix: FILE GENERATION DETAILS 240

B.1 Generated Files 240

B.2 Skeleton Files 240

B.3 IMP Variables 241

1 INTRODUCTION

Toolmake is a facility to help you set up a ToolMaker subsystem, i.e. to create all the necessary files for you in a new directory. This makes it very easy to start a new project based on ToolMaker and you can concentrate on the more essential tasks such as filling the files with your specifications.

Toolmake can generate initial versions of all files needed by ParserMaker, ScannerMaker and ListerMaker. You can choose any combination of these to include in your subsystem. Furthermore ***toolmake*** will offer you an example main module, and if you are using UNIX a **makefile** to build your system.

toolmake is an interactive tool run from the command line. When you use ***toolmake***, it will ask you about the name of your subsystem, and then enter a *command loop*. In this command loop phase you can change the default settings for the coming *generation phase*, or just enter it. You can always give a single question mark (followed by carriage return) in order to get some help. Often you can get further help with a `HELP` command.

2 PRINCIPLES OF OPERATION

When started ***toolmake*** will ask you about the system name and the target language to be used.

Then the command loop is entered which allows you to select or deselect the components you intend to be ToolMaker based. Initially all components are selected. If a component is selected ***toolmake*** will generate template or example versions of the files relating to that component.

The command loop also allows the level of information in the generated files to be specified (minimal, normal or a complete example).

> *Note:* When generating without all components the resulting files will *not* be guaranteed to be compilable without modifications.

Finally the suggestion for a main program and a **makefile** (or command file) may be selected or deselected.

In the generation phase ***toolmake*** produces necessary (explicitly or implicitly requested) description and other files in the current directory by calling the Macro Processor IMP to prepare them from skeleton files in the target language libraries.

3 THE TOOLMAKE COMMAND

Toolmake is invoked by the following call:

```
toolmake [-help] [<option>...] [<system-name>]
```

3.1 Parameters

Toolmake takes one optional parameter, the subsystem name, this parameter is used as default for a subsequent question about the name. Or, as a matter of fact, as the subsystem name in combination with the `-go` option.

3.2 Options

```
-[-]go
```

Do [not] enter the generation phase immediately without starting any command loop (default: off, i.e ask the user first!).

```
-[-]verbose
```

Enable [disable] verbose output mode (default: off). Verbose mode will give some information regarding the actions chosen by ***toolmake***.

```
-help
```

Gives you a brief but informative help on the arguments and options to the ***toolmake*** program.

4 TOOLMAKE RUN-TIME USAGE

When using ***toolmake*** you will normally go through two phases. First specification, when you decide the subsystems name, target language, parts to use etc. Each of the parts may be selected or deselected simply by typing its name, the contents level is selected by typing `level` followed by the level required.

The second phase is the generation phase when ***toolmake*** (by using IMP) actually creates the proper files in your current directory. In this phase you will only need to interact if any errors or unclarities are discovered. The generation phase is automatically entered by giving an empty specification command (i.e. a carriage return without any input).

Toolmake is built to be robust in interaction with the user (that's you), so you can always get help when asked a question by typing a question mark. These properties in addition to the walk-through in *ToolMaker System Description* should give you the aid needed.

4.1 Subsystem Name

The initial question when ***toolmake*** is started is

```
Subsystem name?
```

This question should be answered with the name your new ToolMaker-based system should have. This name will be the basename for all description files, main program template and Makefile-target generated by ***toolmake***.

If the optional argument is used (see section 3.1 on page 233) that is used as the answer for this question and the question is not put.

The specified system name, target language along with the selected components and the contents level is then presented. Then the ***toolmake*** prompt appears.

4.2 ToolMake Commands

At the prompt one of the commands listed below may be used. The default command, which is executed if an empty command is entered, is presented within parenthesis. For example

```
toolmake (GENERATE)>
```

4.2.1 GENERATE

The `generate` command will start the generation of the files (components) currently selected. After completed generation ***toolmake*** is terminated. If any of the generated files already exists the user is required to confirm overwriting them.

4.2.2 QUIT

The `quit` command immediately terminates ***toolmake***, without any generation of files.

4.2.3 ALL

To select all components supported, the `all` command can be used. To select means that they are candidates for being generated when the generation phase is entered.

4.2.4 NONE

The `none` command is the opposite of the `all` command. I.e. all components are deselected.

4.2.5 PARSER, SCANNER, LISTER, MAIN, MAKEFILE

These commands all toggle the selection of the corresponding component. I.e if the component is selected it will be deselected, and vice versa.

4.2.6 LEVEL

The `level` command takes one parameter which is one of

- MINIMAL
- NORMAL
- EXAMPLE

If the level is not specified in the command, ***toolmake*** will prompt for it (together with the default value).

4.2.7 LANGUAGE

The target language to be used in the generated files can be set with the `language` command. It also requires the language as a parameter. Possible languages are

- C
- ANSI-C
- C++

If the language is not specified in the command ***toolmake*** will prompt for it together with the default value in parenthesis.

4.2.8 VERBOSE

The `verbose` command toggles the setting of verbose mode. When ON ***toolmake*** prints some extra information during generation.

4.2.9 HELP

The command `help` will list the available commands together with a short description as opposed to the '?' which only lists the command names.

4.2.10 INFORMATION

The `information` command lists the current setting of the various options, such as selected components, target language and contents level.

A ERROR MESSAGES

The following diagnostic messages may appear when running ***toolmake***. In this list they appear sorted in severity order, with the most severe messages first. *Emphasized* words in the error messages below are substituted with the actual error information.

A.1 System Errors

`System error! malloc: out of memory`

A very serious error since ***toolmake*** hardly uses any dynamic memory, please contact your systems administrator or your ToolMaker contact person.

`System error! function(): switch error (integer)`

A very serious software error in ***toolmake***, please contact your ToolMaker contact person.

`System error! IMP failed to set variable (error code integer)`

`System error! IMP failed (error level integer)`

Internal errors in the communication between ***toolmake*** and ***IMP***, please contact your systems administrator or your ToolMaker contact person.

A.2 Fatal Errors

`Fatal! Language library not found: language name`

The selected language is not yet installed on your system, or the environmental variable TMHOME is not properly set.

`Fatal! Illegal function type: function name`

An error in the command line parsing, please contact your ToolMaker contact person.

`Fatal! Skeleton file halted with error code integer`

IMP executed an %%EXIT command in a skeleton file, this should not occur and should be reported to your ToolMaker contact person.

A.3 Errors

`Error! File not found: file`

Probably the environmental variable **TMHOME** is not properly set, e.g. the indicated directory can contain an older version of ToolMaker.

`Error! Non-matching option: option`

An unknown option was supplied.

`Error! License server: unknown error (number)`

The license file did not have the correct format. Contact your ToolMaker contact person.

A.4 Warnings

`Warning! Argument not used: argument`

Too many arguments where supplied.

`Warning! File exists: file`

This messages informs you that a file to be generated already exists. You are asked a new question about which action to take (quit, rename etc. HELP will give more information).

`Warning! License server: no license available`

You were not able to get a ToolMaker license as there were too many other simultaneous users. Try again later.

`Warning! License server: date expired`

Your ToolMaker license has expired. Contact your ToolMaker contact person.

`Warning! License server: no contact`

ToolMaker was not properly installed or your license server process has died. Contact your ToolMaker contact person or system administrator.

`Warning! License server: date expired`

Your ToolMaker license has probably expired. Please contact your ToolMaker contact person or system administrator.

`Warning! License server: illegal license key`

The password in the license file was illegal. Contact your ToolMaker contact person.

`Warning! License server: format error`

The license file did not have the correct format. Contact your ToolMaker contact person.

`Warning! License server: license file missing`

No license file was found in the **TMHOME** directory. Contact your ToolMaker contact person.

A.5 Informational Messages

Note that informational messages are only printed when the verbose output mode is enabled.

`Information! Generating:` *`file`*

Informational message about the file to be generated.

`Information! Backup file:` *`file`*

If you decide to generate a file although it already exists the old file will be saved with another name.

`Information! Execution stopped`

The previous error was so severe that ***toolmake*** is aborted.

B FILE GENERATION DETAILS

The details in this appendix should be regarded as an internal interface between ***toolmake*** and the skeleton files used, compatibility between two versions of ToolMaker is not guaranteed.

The ***toolmake*** program is however guaranteed to be available in future versions.

B.1 Generated Files

The files generated differs naturally depending on target language, wanted components and generation level. For the default case and a subsystem called x, the following ToolMaker description files will be generated:

- **x.tmk** The common ToolMaker description file.
- **x.smk** The ScannerMaker description file.
- **x.pmk** The ParserMaker description file.
- **x.lmk** The ListerMaker description file.

And for your convenience:

- **x.c** An example main program.
- **Makefile** A suitable file for make(1).

The following source code files will be generated empty by ***toolmake***, to be filled by the various Makers.

- **xCommon.h** subsystem common definitions, e.g. token definition
- **xScSema.c, xScan.c, xScan.h** scanner
- **xPaSema.c, xParse.c, xParse.h** parser
- **xList.c, xList.h** lister
- **xErr.c, xErr.h** error handler, an interface between parser and lister

B.2 Skeleton Files

Toolmake uses skeleton files for all the files it produces. The skeleton files are found in the different language libraries found under the ToolMaker directory, **$TMHOME/lib** (normally **/usr/local/ToolMaker/lib/**).

These files are used as input to IMP together with the appropriate settings of all the variables above. The output should be directed to an appropriate output file in the current directory. For example, for a 'ansi-c'-based system with the system name **pl0**:

```
imp -s_T("12:34:56") -s_D("1990-01-12") \
    -s_sN("pl0") -s_tL("ansi-c") ... \
    $TMHOME/lib/c/main.imp pl0.c
```

For more details on using the Macro Processor IMP refer to the *IMP Reference Manual.*

The skeleton files for the files produced by ***toolmake*** are:

- **pmk.imp** ParserMaker description file skeleton
- **smk.imp** ScannerMaker description file skeleton
- **lmk.imp** ListerMaker description file skeleton
- **tmk.imp** common description file skeleton
- **main.imp** main program skeleton
- **makefile.imp** makefile skeleton
- **voc.imp** ScannerMaker vocabulary file skeleton

B.3 IMP Variables

Toolmake uses SoftLabs Incremental Macro Processor, IMP, in order to generate the files properly. In each call to IMP, ***toolmake*** will set some IMP variables. These are used by the IMP scripts to produce the appropriate file. The following variables are required:

Variable	**Description** (example value)
_T	Current time ("23:59:59")
_D	Current date ("1999-12-31")
_P	Producers name ("toolmake")
_V	Version information (v2,r0,c0)
_sN	System name ("pl0")
_Lvl	Contents level, MINIMAL, NORMAL or EXAMPLE (NORMAL)
_tOS	Target OS (SunOS)
_tL	Target language ("ansi-c")
_tD	Target directory below $TMHOME/lib ("ansi-c")
_tX	Target language file extension ("c")
_i	Included parts of ToolMaker (sm, pm, lm, tm)
_mm	If main module generated (YES)

If you would do a manual call to IMP in order to produce any of the library files, see below, you must supply these variables. This should however be avoided since this may not be compatible with coming versions of ToolMaker.

Index

Symbols

#line 91

A

acceptance cost 73
ambiguous grammar 68

B

BNF 60

C

character class 156
character map 149
character set 142, 148
closure 155
common options 30
    relevant for ListerMaker 202
    relevant for ParserMaker 87
    relevant for ScannerMaker 141
context 164
cut operator 155

D

delete cost 95
description file 25
disambiguating rules 68, 71

E

EBNF 60
embedded semantic actions 67
end of input 157
end of text 157
error correction 72
error message 22
error recovery 72
    improvement 94
    interface module 108
    tuning 75

F

fiducial symbol 74, 75, 98
full list 213

G

global messages 198
grammar attributes 64
    in EBNF rules 65
grammar file 77
grammar production 62

H

hook 147

I

IMP 24
include 210, 213
Incremental Macro Processor 24
inherited attributes 67
insert cost 95
insert string 199, 218
insertion marker 199, 218
item set 100

L

LALR 23, 68
left recursive grammar 89
lexical attributes 67
Lister 198
ListerMaker 24
listing file 27, 28, 100
listing information 200
listing phases 209

M

Maker 25
Maker prefix 42
markers 28, 199, 218
message
    number 198, 219
    template 28, 198, 201, 218
message sections 204
metalingustic variables 60
modification rules 71

N

nested files 210, 213
non-terminals 60

O

option format
    in description files 40
    on the command line 40
option precedence 42

P

packing 90, 143
panic mode 75
parse stack overflow 89
parser entry point 107
parser prefix 76
ParserMaker 23
posthook 147
prefix 42, 76, 136
prehook 147

R

reader 146
recovery point 107
reduce-reduce conflict 68
regular expressions 154–158
right recursive grammar 89

S

scanner 134
scanner context 164
ScannerMaker 24
screening 134, 143
semantic actions 63, 66, 99, 134
    embedded 67
severity 199
shift-reduce conflict 68
single symbol correction 73
skeleton files 26, 77, 137
source position 22, 198
source position calculation 135, 166
standard error 214
standard input 146
string synthesising 74
syntax error 72
synthesized attributes 66
system name 234
system prefix 42, 76, 97

T

table packing 90, 143
terminals 60
tiny list 213
token 134
token buffer 144
token type 36
toolmake 24, 29
ToolMaker Common Description
        file 26, 100, 160, 204

U

unknown token 158

V

vocabulary 28, 103, 134, 151
vocabulary file 27, 102