Draco 1.2 Users Manual

21 June 1983

by James M. Neighbors

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

This work was supported by the National Science Foundation Software Engineering Section under NSF grant MCS-81-03718 and by the Air Force Office of Scientific Research.

Table of Contents

CHAPTER 1
AN INTRODUCTION TO THE DRACO SYSTEM

It has been a common practice to name new computer langauges after stars. Since the system described in this manual is a mechanism which manipulates special purpose languages it seems only fitting to name it after a structure of stars, a galaxy. Draco [NOTE 1] is a dwarf elliptical galaxy in our local group of galaxies which is domainated by the large spiral galaxies Milky Way and Andromeda. Draco is a small nearby companion of the Milky Way (1.2E+5 solar masses and 68 kiloparsecs from Earth). This small size and close distance to home is well suited to the current system which is a small prototype.

1.1. The Draco View of Software Production

The Draco system addresses itself to the routine production of many systems which are similar to each other. The theory behind its operation is described in detail in [Neighbors80].

Three themes dominate the way Draco operates: the use of special-purpose high-level languages for the domains or problem areas in which many similar systems are needed; the use of software components to implement problems stated in these languages in a flexible and reliable way; and the use of source-to-source program transformations to tailor the components to their use in a specific context. The basic steps in the production of a specific system using a Draco supported domain-specific high-level language is as follows:

1. An analyst with experience in developing many systems in a certain problem domain decides that the domain is understood well enough to define a language suitable for comfortably and easily describing other systems in the problem domain. This person is called the **Domain Analyst** and the language described is called the **Domain Language**. The Domain Analyst describes the domain and its internal form with the parser generator part of the BUILD subsystem of Draco which is described in Chapter 2.
2. Once the Domain Analyst has described the external and internal form of the domain then how program fragments in the domain should be printed so that users find them easy to look at and accurate in their meaning must be described. This is called prettyprinter generation and it is done by the Draco BUILD subsystem using the notations described in Chapter 3.
3. The Domain Analyst must provide simplifying relations among the objects and operations of the domain. These are used for simplification and optimization of programs in the domain. These simplifications are accepted in terms of source-to-source program transformations by the BUILD subsystem which forms them into a library of transformations. The creation of transformations is discussed in Chapter 4.
4. Finally, the Domain Analyst must prepare a prose description of the meaning of the operations and objects in the domain.
5. This prose description is turned over to a **Domain Designer** who specifies components for the objects and operations in the domain which refine the objects and operations of one domain into other domains known to the Draco system. These components are formed into libraries by the Draco subsystem BUILD from specifications described in Chapter 5. A component is a set of refinements each capable of implementing a domain object or operation under certain stated conditions while making certain implementation assertions.
6. A new system defined by a **Systems Analyst** which can be described in a Domain Language known to Draco can inherit some analysis, design, and coding from the Draco library. The statement of the system to be constructed is cast in a Domain Language. The Domain Language program is then turned into an internal form by the PARSE subsystem. The use of the PARSE subsystem is described in Chapter 6. This internal form is then given to a System Specialist.
7. The **System Specialist** interacts with the transformation and refinement subsystem of Draco called TFMREF (described in Chapter 7). The basic operation in this phase is the selection of an appropriate set of software components to implement the operations and objects in the domain which are used in the problem statement. Then these components are specialized by program transformation (described in Chapter 8) to the problem at hand and then separately refined (described in Chapter 9) into another (or the same) domain and the cycle begins again. The TFMREF subsystem allows the definition of refinement tactics (described in Chapter 10) capable of removing the burden of answering low-level questions from the System Specialist.
8. The process the System Specialist uses to refine the problem is, of course, not strictly top down but the TFMREF subsystem keeps a record of the process which makes it look top down. When the program is in an executable form it is printed out by the System Specialist and either acceptable or the specification cycle begins again with the existing Domain Language program.
9. The refinement history of a program may be examined by a user of the EXAMINE subsystem [NOTE 7] which states what refinements were used in the production of this program. A higher-level description of all parts of the program to whatever level (up to the orginal Domain Language) always exists in the refinement history. It is hoped that these higher levels of abstraction of an existing program will be useful in understanding the program during the maintenance phase of its lifecycle.

The process described briefly above is dealt with in more detail in [Neighbors80] which presents an SADT [NOTE 2] model of the process.

1.2. Running the Draco System

This section describes the loading and execution of the Draco system on the ICS PDP20A at U.C. Irvine as of June 1, 1983. At this time access to the Draco system is restricted to members of the Reusable Software Project at U.C. Irvine [NOTE 3] . As in all the example transcripts in this manual the user input is **strong** type and terminated with RETURN and comments are in *{ephasis}* type.

```
{we enter at the monitor level on the PDP20}
@DEFINE DRACO: <DRACO> {the Draco disk area}
@DRACO
{screen clears}
Draco 1.2
{some notices and bug messages are printed here}
DRACO>HELP
{the current legal Draco commands are printed}
The Draco commands are:

 BUILD  - generate a domain language parser (.DEF->.PAR)
        - generate a domain language internal form prettyprint
        - generate a domain transformation library (.TFM->.TLB
        - generate a domain refinement library (.REF->.RLB)
 PARSE  - parse a program into internal form ( ?->.INT)
 TFMREF - transform and refine a program (.INT,.HIS->.INT,.HIS
 EXAMINE- examine the refinement of a program (.HIS)
 SET    - set terminal type and other environmental parameters
 EXIT   - return to the monitor level
 LISP   - reenter LISP
 HELP   - this listing

A "^" to a filename or (Y/N) request exits current subsystem.
DRACO.INI initialization file may contain any of these command
```

The rest of the sections of the manual assume that Draco is loaded and already in execution.

1.3. Interacting with Draco Menus

Draco uses a standard menu interaction which includes command completion and a help facility [NOTE 4]. The following keys control the menus in Draco.

- RETURN terminates commands and requests execution.
- LINEFEED requests information about the options which exist at the current point in the menu.
- ESCAPE requests that the menu driver fill in the current choice if it is unambiguous and prompt for the next menu item required. If the current command is still ambiguous then the terminal will beep and the cursor will not move. At this point a RETURN will abort the command because it is ambiguous and a LINEFEED will list the acceptable inputs.

Once one of the activation characters given above has been given no characters before the activating character can be deleted. This is an effect of only activating on these characters and not activating on RUBOUT, DELETE, or BACKSPACE. A command which has been entered correctly is usually aborted with a RETURN. The following error messages are given by the menu driver when a command is aborted.

- **?Incomplete Command** is issued when all the fields required by the command have not been filled in and a RETURN was given to activate the command.
- **?Ambiguous Command** is issued when a RETURN was given to activate the command and either a sub-command is needed or the specification of the original command does not contain enough characters to differentiate between a group of commands. LINEFEED at the same point will show the possible commands needed to complete.
- **?Unknown Command** is issued when a RETURN was given to activate the command and the given command is not one of the possible choices.
- **?Command Unconfirmed** is issued when all the fields of a command have been filled in but the last field was not terminated by a RETURN. The assumption is that a user does not understand the command if he prompts for more fields on the last field.
- **? <fieldname> was not specified** is issued when some but not all of the required fields of a command have been specified. The fieldname given is the next field to be entered.

Usually menus contain a "help" command which will print out detailed documentation for the commands in the menu. The "^" command will exit the menu returning to the next higher subsystem. In addition a "^" command to most requests for an input file name will return to the next higher menu.

1.4. Terminal Definition for Draco

Some of the features available in new terminals are used by Draco to highlight information and interact with the user. Primarily these interactions are based around the ANSI terminal standards.

Some flexability in defining new terminals is described in appendix APPX-TERMINAL. Commands from the main menu of Draco (i.e., commands acceptable at the DRACO> menu prompt) can be put into a command file entitled DRACO. INI. This is useful for setting up terminal types and getting updates of the Draco software.

1.5 Overview of the User Manual

The manual is organized in 10 chapters and 10 appendicies. The chapters are organized as follows:

Chapter 1	Introduction to the Draco approach and general guide to the system.
Chapter 2	Detaled explanation of how to write parser definitions using Draco and how to build a parser for a domain language.
Chapter 3	Detailed explanation of how to write a prettyprinter definition and how to build a prettyprinter for a domain language.
Chapter 4	Explanation of how to write transformations and how to build a transformation library for a domain language.
Chapter 5	Explanation of how to write components with refinements and how to build a component library.
Chapter 6	Explain and give example of how to parse a program in a domain language to the Draco internal form.
Chapter 7	Explain the use and the main commands of the Transformation and Refinement subsystem of Draco.
Chapter 8	Explain how a transformation library should be used to optimize a given program.
Chapter 9	Explain how a System Specialist should use the system to to refine a program written in a domain language into an executable language. It describes all the necessary commands to do a refinement.
Chapter 10	Explain the Tactics subsystem and show how to use it.
Appendix I	Gives a complete example of a domain language (SPL).
Appendix II	Gives the definition of the main parser generator, the Draco parser writen in Draco.
Appendix III	Gives the difinition of the Draco prettyprinter domain.
Appendix IV	Gives an example of a prettyprinter description (SPL).
Appendix V	Gives an example of transformations.
Appendix VI	Gives the definition of the Draco Component Library domain language, the language used in describing components.
Appendix VII	Gives the definition of the interpreter of the Tactics subsystem used in guiding refinement.
Appendix VIII	Gives an example of a terminal type to Draco.
Appendix IX	Gives the definition of the Tactics prettyprinter.
Appendix X	Draco errors and messages.

CHAPTER 2
DESCRIBING A DOMAIN LANGUAGE

Once the analysis of a problem domain has been completed the Domain Analyst must define a language suitable for describing solutions to programming problems in the domain. This high-level language should be very specific to the domain and capable of describing the objects and operations of the domain in a comfortable way.

In this section we are concerned with how to specify the external form (syntax) and internal form of a domain language to the Draco subsystem BUILD. The chapters on transformations and refinements are concerned with specifying the semantics of the language.

2.1. The External Form (Syntax) Specification

Classically BNF has been used to describe the syntax of languages and Draco carries on this tradition. The Draco BNF is similar to the BNF used in syntax directed compiling which is the foundation of the META systems [Schorre64].

2.1.1. Draco BNF Described in BNF

The Draco BNF is described below by standard BNF with the following metasymbols: <name> denotes a rule, {obj} denotes zero or more occurrences of obj, | denotes alternation , and a single word or character with quotes on either side denotes itself. Where <character> matches any character and <schar> matches any character except a double quote (").

```
<DracoBNF>::= .DEFINE <identifier> {<Draco-rule>} .END
<Draco-rule>::= <parse-rule> ; | <token-rule> ;
<parse-rule>::= <identifier> = <parse-exp>
<parse-exp>::= <parse-seq> { / <parse-seq>} |
               <parse-seq> { | <parse-seq>}
<parse-seq>::= <parse-ele> {<parse-ele>}
<parse-ele>::= <identifier> | <string> | ( <parse-exp> ) |
               <parse-iteration> | .EMPTY |
               [ [ <parse-exp> ] <parse-exp> ]
<parse-iteration>::= $ < <iteration-range> > <parse-ele> |
                     $ <parse-ele>
<iteration-range>::= <iteration-number> : <iteration-number>
<iteration-number>::= <number> | ?
<token-rule>::= <identifier> : <token-exp>
<token-exp>::= <token-seq> { / <token-seq>}
<token-seq>::= <token-ele> {<token-ele>}
<token-ele>::= <identifier> | <char-rule> | ( <token-exp> ) |
               <token-iteration> | .EMPTY
<token-iteration>::= $ < <iteration-range> > <token-ele> |
                     $ <token-ele>
<char-rule>::= .ANY ( <char-exp> ) | .ANYBUT ( <char-exp> )
<char-exp>::= <char-range> { ! <char-range>}
<char-range>::= <char-value> | <char-value> : <char-value>
<char-value>::= <number> | ' <character>
<identifier>::= <alphabetic> |
                <alphabetic> {<digit>} {<identifier>}
<number>::= <digit> {<digit>}
<string>::= " <schar> "
<alphabetic>::= A | B | ... | Z | a | b | ... | z
<digit>::= 0 | 1 | ... | 9
```

Figure 2-1: BNF for Parser Definition

2.1.2. An Example of Draco BNF

The <token-rule>'s specify how to collect characters into tokens (lexemes) while the <parse-rule>'s specify how to group tokens together to parse the external form. The <char-rule>'s specify what characters to accept within a <token-rule>. The iteration rules (<parse-iteration> and <token-iteration>) are similar to the {} notation used above and specify sequences which may occur zero or more (up to an optional limit) times. Kleene * and + are a subset of the available values of iteration.

As an example of the Draco BNF consider the following which describes simple parenthesized arithmetic ALGOL-like assignment statements.

```
    .DEFINE ASGN

    ASGN = IDENTIFIER ":=" (EX1 / STRING) ";" ;
    EX1  = EX2 $("+" EX2) ;
    EX2  = EX3 $("*" EX3) ;
    EX3  = EX4 $("^" EX3) ;
    EX4  = IDENTIFIER ( "(" EX1 $("," EX1) ")" / .EMPTY) /
           NUMBER / "(" EX1 ")" ;

    PREFIX : SPACES ;
    IDENTIFIER : SPACES ALPHA $<?:5>(ALPHA / DIGIT) ;
    NUMBER : SPACES DIGIT $DIGIT ;
    STRING : SPACES .ANY('") $.ANYBUT('") .ANY('") ;
    ALPHA : .ANY('A:'Z ! 'a:'z) ;
    DIGIT : .ANY('0:'9) ;
    SPACES : $.ANY(32) ;

    .END
```

The .DEFINE tells Draco that this is a domain language description and the name which follows is the name of the first rule to be invoked. Sequences enclosed in double quotes (") are literal strings which are tested to see if they appear (without double quotes) in the input stream. The slash (/) denotes alternation similar to the logical bar (|) of BNF. So, an ASGN is started by an IDENTIFIER which must be followed by the sequence ":=" which is then followed by either a sequence described by rule EX1 or a STRING. Whichever alternative is elected it must be followed by a semicolon (;).

IDENTIFIER is a <token-rule> and it scans off individual characters. An IDENTIFIER is a sequence of zero or more spaces (32 is the decimal ASCII representation of a space) followed by an upper or lower case letter of the alphabet followed by zero to five letters or digits. A STRING is a sequence of zero or more spaces followed by a double quote, followed by the string characters (any chararacter except a double quote), followed by a double quote. The following are legal ASGN statements according to the Draco BNF above.

```
PHI := (col7 + col5)*FUDGE ;
Person := "Edward the Great" ;
VAL7 := 5+3*6^4 ;
ITS := ((A+6)*3)+7+6+5^power ;
Zee := factor*SIN(2*Pi) ;
APE := FURD(5,FURD(3,B)) ;
```

2.1.3. The PREFIX and SUFFIX Rules

Two rule names, PREFIX and SUFFIX, are used to shorten the Draco BNF description of a langauge. If a Draco BNF description contains a PREFIX rule then this rule is applied before every test for a literal string (characters enclosed in quotes). Thus,

```
ASGN = IDENTIFIER ":=" (EX1 / STRING) ";" ;
PREFIX : SPACES ;
```

is the same as

```
ASGN = IDENTIFIER SPACES ":=" (EX1 / STRING) SPACES ";" ;
```

The SUFFIX rule operates in a similar manner except that if it exists it is applied after the test for the literal string has been successful. If the PREFIX rule didn't exist in the example above, then the statement

```
PHI:=( col7+ col5)* FUDGE;
```

would be legal while

```
PHI := (col7 + col5)*FUDGE ;
```

would not be legal because of the embedded spaces. In general the PREFIX and SUFFIX rules are useful in shortening the description of languages without fixed fields.

2.1.4. Controlling Parser Backtracking

The alternation (/) used in Draco assumes that one of the alternatives will succeed in matching the input stream. A sequence succeeds in matching the input stream if all of the objects indicated in the sequence are found in the input stream (remember the sequence operator is a blank). If the first object in the sequence is not found then the sequence operator indicates a failure to recognize. If the first object in the sequence is found but some other part of the sequence is not found then a problem occurs in that the pointer into the input stream has been advanced over the first object. In this case the sequence operator indicates a syntax error but does not report it to the user yet. The alternation operator (/) passes the syntax error on up to the construct above it.

The backtracking operator (|) traps the syntax error returned by a nested sequence operator, restores the state of the parser to the point where the backtracking operator was entered, and trys the next alternative. In short a backtracking operator is the same as an alternatation operator except that the state of the parser is saved and restored between the alternatives. The backtacking operator indicates failure to recognize if none of the alternatives are present in the input stream. The backtracking operator never results in a syntax error indication. The backtracking operator is more expensive in time and space because of its state saving and restoring. One could use only backtracking operators in a parser definition without any alternation operators but the resulting parser would be very slow in execution. The basic idea of having the two similar operators is to be able to specify in a parser description where the language described is simple LL(1) parseable and where LR(k) parsing must be used.

As an example of where backtracking is needed consider the following Draco BNF description.

```
A = B | "a" "f" "g" ;
```

```
     B = "a" "f" "h" / "a" "f" "i" ;
```

The strings "afg" and "afh" are recognized by the grammar but the string "afi" would result in a failure of the rule A without advancing the input pointer. If given an "afi" in the input stream the rule B would recognize the "a" and "f" in the first alternative and issue a syntax error to the backtracking operator in the A rule because the first two elements of the sequence were present but the "h" was missing. The other alternative is not even tried. The "afg" is recognized because the B rule returns a syntax error to the backtrack in A which restores the parser input pointer to be pointing to the "a" and then tries its next alternative. We could rewrite the grammar in two ways: by replacing the alternative in the B rule by a backtrack

```
     A = B | "a" "f" "g" ;
     B = "a" "f" "h" | "a" "f" "i" ;
```

or by factoring the alternative in the B rule.

```
     A = B | "a" "f" "g" ;
     B = "a" "f" ("h" / "i") ;
```

The second option is of course faster in execution but the main issue in writing parser descriptions is to make the grammar clear.

2.1.5. Error Recovery During Parsing

If the only control constructs we used in parser descriptions were sequence, alternation, and backtracking then the error recovery power of the parsers would be severly limited. By error recovery here we mean being able to handle ill-formed statements in the language, report them to the user, pass over them in the input stream, and continue parsing.

Once a sequence operator reports a syntax error all alternation operators will pass on the error and backtracking operators will trap the error and try their next alternative. A simple ill-formed expression would usually cause the entire parse to fail either by backtracking out of the top level rule or by passing a syntax error back from the top level rule which would abort the parse.

Some error control could be built in by using the backtracking operator but we have decided to introduce a special error recovery mechanism called an error block. The syntax of an error block is as follows.

```
                    [[ <parse-ele> ] <parse-ele> ]
```

The first expression of the block is attempted and if a syntax error results then the state of the parser is restored to the point where the error block was encountered, an error message is printed indicating the rule which originated the error and the position in the input stream at the time of the error, finally the second expression is attempted. It is the goal of the second expression to skip over the ill-formed statement. If the second expression results in an error then the user is again notified that error recovery was unsuccessful and the syntax error is returned as the result of the error block with the state of the parser restored to the point where the error block was encountered. If the first expression in the error block succeeds or fails then it is the result of the error block. The error block only stops syntax errors.

As an example of using an error block consider the following grammar which recognizes statements (STMT) followed by a semicolon. The error recovery strategy for an ill-formed STMT is to scan all the characters up a semicolon. The error reporting is already handled by the error block.

```
     BODY = $[[ STMT ";" ] STERR ";" ]
     STERR : .TOKEN $.ANYBUT(';) .DELTOK ;
```

If a syntax error occurs inside of the error recovery part of an error block then a message is given that the error recovery has failed is given and the syntax error propagates out of the error block. It is important for the parser designer to remember that an error message is printed to the user *every* time a syntax error occurs. Thus syntax errors should not be used by a parser designer as a control strategy.

Token rules (indicated by a : rather than a =) never generate a syntax error and never advance the input pointer on a failure. The token buffer always contains the token recognized in the input by the last token which succeeded. The manipulation of the token buffer will be described in a later section on internal forms.

Sometimes it is useful to be able to explicitly control the issuing syantx-error and rule-failure conditions in the parser. This can be done using the .FAIL and .ERROR constructs. As can be guessed, the .FAIL construct fails the current parse rule immediately, without regard for any alternatives, sequences, backtracks, or errorblocks in which it is embedded. The .ERROR construct raises a syntax error when it is encountered, and it is dealt with in a manner similar to other synatx errors.

2.1.6. Elements of External Form Description

This section summarizes the external form description mechanisms in Draco BNF. In both <parse-rule> and <token-rule>

Syntax	Meaning
A B ...	sequence - an A followed by a B followed by ...
A|B|...	backtrack - A or backtrack B ...
A/B/...	alternation - an A or a B or ...
.EMPTY	the last element of a alternation states that none of the alternatives need be taken
[[A]B]	error block - try A and B handles errors
(A)	encapsulation - treat as one unit
$A	iteration - zero or more instances of A
$<n:m>A	iteration - n to m instances of A (? implys any number)

In <token-rule> only

Syntax	Meaning
.ANY(A)	scan any char described by A
.ANYBUT(A)	scan any char not described by A

Inside .ANY or .ANYBUT character class descriptions

Syntax	Meaning
'A	characters equal to the ASCII value of A (65)
65	characters equal to the ASCII value of 65 (A)
A!B!...	characters matching A or B or ...
A:B	characters whose ASCII value C is such that A<=C<=B

The precedence of the parser control contructs is as follows:

Rank	Operator	Symbol
Highest	encapsulation	()
	sequence	space
	backtrack	|
Lowest	alternation	/

2.1.7. Recognizing the End of the File

Some languages do not have explicit end of input markers (such as END statements) so Draco has a facility enabling domain language parsers to recognize the end of an input file. When Draco recognizes the end of the input file it places *one* control-Z (ASCII 26) in the input stream to be recognized by the parser. If the parser does not recognize the control-Z and it tries to read further then an error will occur.

2.2. The Complete External/Internal Form Specification

The Draco system expects the internal representation of a program to be a tree. Each node in the tree must have an identifying name as the first entry in the node. This form is called prefix form. As an example the fragment 5*A+B+C^7 could be represented internally as

```
 .----+--------------------+
 !    !                    !
ADD   .----+------------+  .----+----+
      !    !            !  !    !    !
     ADD   .----+----+  B  EXP  C    7
           !    !    !
          MPY   5    A
```

which is a legal prefix form in that the leftmost entry in each node is a name (the prefix keyword). Each node has a fixed number of entries. All nodes with the same prefix keyword have the same number of entries.

These prefix form trees are built from the bottom up as Draco scans a program in a domain language. In particular when a token is recognized in a <token-rule> it is stored in a token buffer. It is then the responsibility of a <parse-rule> to take the token, combine it into a new node, and insert it into the growing tree. The growing tree is maintained as a stack of objects which have not yet been combined into higher nodes. The prefix form for a domain should have a single root which is left as the last node on the stack.

2.2.1. External/Internal Form Specification

The operators for constructing internal forms are mixed in with the Draco BNF notation and each is preceeded by a period (.) . The internal form construction operators should not be confused with .ANY and .EMPTY which are part of the external form specification. Only two rules from the earlier BNF specification of the syntax need be changed to add the tree construction operators. The two revised rules and a new rule are given below.

```
<parse-ele>::= <identifier> | <string> | ( <parse-exp> )
               <parse-iteration> | .EMPTY | .LITERAL | .LITCHAR |
               .NODE ( <identifier> {<node-ele>} ) |
               .TREE ( <identifier> <identifier> <parse-exp> ) |
               .CHART ( ???? )
<token-ele>::= <identifier> | <char-rule> | ( <token-exp> ) |
               <token-iteration> | .EMPTY | .TOKEN | .DELTOK
```

2.2.2. Specifying a Legal Parser

Some restrictions exist as to what a parse rule may add to the stack of nodes which construct the internal form tree.

1. If a parse rule succeeds it can only put one node in the node stack. Multiple nodes may be constructed during the parse rule (constructing subtrees) but when the rule succeeds the net change in the number of nodes in the node stack can be only one. This rule makes sure that the internal from returned by a nonterminal in the syntax (a parse rule) is always a tree with the single node returned being the root. This concept will be used later when we discuss describing software components.
2. If a parse rule fails it may not add any nodes to the node stack.

Remember that the goal of parsing for Draco is to produce an internal form which captures *all* the information in the syntax of the problem domain. The one parse rule one node restriction we have found guides to parser designer in capturing the entire syntax of the domin in the internal form.

2.2.3. A Complete External/Internal Form Example

We will redo our assignment statement example from a previous section adding the internal form construction information.

```
    .DEFINE ASGN

    ASGN = IDENTIFIER .LITERAL
           ":=" (EX1 / STRING .LITERAL)
           ";" .NODE(ASSIGN #2 #1) ;
    EX1  = EX2 $("+" EX2 .NODE(ADD #2 #1)) ;
    EX2  = EX3 $("*" EX3 .NODE(MPY #2 #1)) ;
    EX3  = EX4 $("^" EX3 .NODE(EXP #2 #1)) ;
    EX4  = IDENTIFIER .LITERAL
            ( "(" APARAMS ")" .NODE(FNCALL #2 #1) /
              .EMPTY ) /
           NUMBER .LITERAL /
           "(" EX1 ")" ;
    APARAMS = .TREE(APARAMS APSEQ EXP $("," EXP)) ;

    PREFIX : SPACES ;
    IDENTIFIER : SPACES .TOKEN ALPHA $<?:5>(ALPHA / DIGIT) .DELTOK
    NUMBER : SPACES .TOKEN DIGIT $DIGIT .DELTOK ;
    STRING : SPACES .TOKEN .ANY('") $.ANYBUT('") .ANY('") .DELTOK
    ALPHA : .ANY('A:'Z ! 'a:'z) ;
    DIGIT : .ANY('0:'9) ;
    SPACES : $.ANY(32) ;

    .END
```

First of all, notice the .TOKEN and .DELTOK operations which have been added to the token rules. The .TOKEN states which character should be the first in the token and the .DELTOK places the token in the token buffer . In this case the .TOKEN in the IDENTIFIER rule states that the initial spaces are not part of the identifier.

The <parse-rule>'s form internal tree nodes from a stack of objects. The operator .LITERAL takes the last token put into the token buffer by a .DELTOK and pushes it on the stack. The operator .NODE( ) takes objects from the stack (#), the token buffer (*), and literal data (ADD, MPY etc.), forms these into a new node and pushes it on the stack. Thus, .NODE(*) takes just the token buffer, forms it into a node, and pushes it on the stack. The operation .NODE(ADD #2 #1) forms a new node consisting of the literal ADD, the second object on the stack, and the object on top of the stack. The operation .NODE(MPY #1 #1) forms the literal MPY and the top two elements of the stack into a node.

WARNING

Note that this is tricky, .NODE(EXP #1 #2) forms the top of the stack and the ***third*** *element of the original stack into a new node. The # operation removes the elements from the stack when they are fetched.*

2.2.4. An Example Internal Form

Using our example assignment definition above the prefix internal form of the statement

```
                       ANS:=GEO(B,2*E)+E^2^C ;
                                  is
.-------+---+
!       !   !
ASSIGN  ANS .---+---------------------+
            !   !                     !
            ADD .------+---+          .---+----+
                !      !   !          !   !    !
                FNCALL GEO .-------+  EXP E    .---+--+
                           !       !           !   !  !
                           APARAMS .------+--+ EXP 2 C
                                   !      !  !
                                   APARAM B  .------+--------+
                                             !      !        !
                                            APARAM  .---+--+ *OMEGA*
                                                    !   !  !
                                                    MPY 2  E
```

Notice that the precedence of the operators is done by ordering the <parse-rule>'s and that * (multiply) is left-associative while ^ (exponentiation) is right-associative.

2.2.5. Variable Length Structures in Internal Forms

Due to the restriction that all nodes of a certain type have the same number of subtrees, some mechanism must be developed to allow a variable number of elements in some cases. For example, not all programs have the same number of statements in them, so some structure must be developed to hold a variable number of statements. In Draco this is done by means of right-leaning trees with header nodes, internal nodes, and a special termination marker. For example, the set of program statements could be represented internally as:

```
.------+
STMTS  !
       .---------+------+
       !         !      !
       STMT-SEQ  stmt1  .---------+------+
                        !         !      !
                        STMT-SEQ  stmt2  .
                                         .
                                         .
                                         .---------+------+
                                         !         !      !
                                         STMT-SEQ  stmtn  *OMEGA*
```

If we had a parse rule GET-STMT which could build nodes for statements in a particular language, then the construction of this internal form could be achieved by the syntax

```
               .TREE(STMTS STMT-SEQ $GET-STMT)
```

The .TREE construct always takes the three arguments: the header node name for the tree, the internal node name for the tree, and an expression which produces multiple nodes to be linked together in the tree. This internal-form structure is known and exapected by the refinement part of Draco. It is acceptable to have a tree with no internal nodes indicating a variable length structure with no elements.

While the .TREE constructor is used for scanning variable-length structures from "Top to bottom" and building a tree, some mechanism must be defined for scanning sets of variable length structures from "left to right." An example of such a structure is a table in which we wish to associate the columns together in a tree rather than the rows, even though we must scan through the table a row at a time. Consider the problem where we wish to scan the following table of data:

```
female Sally
male   Dick
female Jane
```

into the following internal form

```
.---+-----------------+
!   !                 !
SET .---+             .-----+
    !   !             !     !
    SEX .-----+-+     NAMES .-----+-----+
        !     ! !           !     !     !
        S-SEQ F .-----+-+   N-SEQ Sally .-----+----+
                !     ! !               !     !    !
                S-SEQ M .-----+-+       N-SEQ Dick .-----+----+
                        !     ! !                  !     !    !
                        S-SEQ F *OMEGA*            N-SEQ Jane *OMEGA*
```

Given the parse rules GET-NAME and GET-SEX which could produce the appropriate nodes, this internal form could be constructed by the fragement

```
       .CHART(SEX S-SEQ NAMES N-SEQ $(GET-SEX GET-NAME)) .NODE(SET #1 #1)
```

The .CHART construct accepts a variable number of header-nodes and internal-node pairs followed by an expression to produce nodes. The number of nodes produced by the expression before it fails must be an even multiple of the number of node name pairs.

2.2.6. Elements of Internal Form Description

This section summarizes the internal form mechanisms in Draco BNF. In <token-rule>'s only

Syntax	Meaning
.TOKEN	show the start of the token
.DELTOK	put the token into the token buffer

In <group-rule>'s only

Syntax	Meaning
.LITERAL	push the token buffer on the stack
.NODE( )	form a new node and push it on the stack
.LITCHAR	push the ASCII value of the next character on the stack
.TREE(A B E)	evaluate E until fail and build right leaning tree with A top node, B internal nodes (possibly none), and *OMEGA* terminator of the sequence (vertical parsing)
.CHART(A B ... C D E)	evaluate E until fail and then build n number of right leaning trees with A,..,C as top nodes and B,..,D as internal nodes, and *OMEGA* as terminator of all trees (horizontal parsing)

Inside of a .NODE( ) only

Syntax	Meaning
<identifier>	literal data
<number>	literal data
# <number>	pop nth object on stack and use as is

2.3. Special Functions in Parsers

In this section we will discuss three major features available to parser builders which do not affect the syntax of the language. In particular, we will discuss data-flow-consistency checking functions, diagnostics to the user, and nonstandard, internal-form constructors.

2.3.1. Checking Consistency in Parsers

Within a parser for a certain language, it is nice to be able to check the consistency of the objects given in the user's progarm. For example, if the language allows function calls, and if a function is called in the users's program, the parser should ensure that the function is defined later. Quivalently, if a function is defined then the parser should ensure that some other part of the program is using it. Within Draco, these operations are carried out by the following parser constructs:

Syntax	**Meaning**
.DEF(type)	The .DEF construct declares that the contents of the token buffer contain the identifier of an object which is defined to be of the given type. The type is just a name made up by the parser builder. FUNCTION would suffice for the example given above.
.USE(type)	The .USE construct declares that the contents of the token buffer contain the identifier of an object which has been referenced as the given type.
.RESOLVE (type)	The .RESOLVE construct checks to see that all the objects of the given type which have been defined have been referenced. It also checks that all the objects of the given type which have been referenced have been defined. Error messages will be printed if any discrepancies occur. However, no syntax error or failure will be issued from the construct.
.RETRACT (type)	The .RETRACT construct erases any .USE's or .DEF's for the given type in the current context.
.CONTEXT-PUSH(type)	The .CONTEXT-PUSH construct saves all .DEF's and .USE's for the given type on a stack and erases them in the current context. This is useful for objects with nested scoping such as labels local to BEGIN-END blocks. Upon entering the block the lables are pushed, and upon exiting the block the labels are first resolved, and then popped.
.CONTEXT-POP(type)	The .CONTEXT-POP construct retrieves the definitions for the type previously saved on the stack. The stack is not the same as the stack used in constructing trees.
.ASSUME (type)	The .ASSUME construct can be used to assume declarations for objects of the type which have been .USEd and not .DEFed. Each time the .ASSUME construct is referenced, it will either result in a fail (which means that there are no more objects of the type to be assumed), or it will result in syntax recognition with the identifier of the next object of the type to be assumed put on the node stack and automatically DEFed. For example, this is useful in the declaration of local variables in a function. All variables used in the function, and not declared to be global, could be assumed to be local without having to have both a local and a global declaration.

2.3.2. Notifying the User

While the parser is parsing the user's program, it is nidce to be able to tell the user what is going on. For example, it is nice to tell the user what major part of the program is currently being parsed. This is doen with the .MSG construct. Within the .MSG construct the following items are acceptable:

Syntax	**Meaning**
"abced"	A string to be printed.
*	Print the token buffer contents.
.CR	Print a carriage return and linefeed.
.COL(value)	Advance the carriage to the given column.

Each time a "crlf" is encountered, "*" markers are printed as the parser does its work. Most parser messages will need a .CR first.

2.3.3. Non-Standard Parser Constructs

The fillowing parser constructs are briefly described in the interest of completeness, but they should not be used by domain parser builders:

Syntax	**Meaning**
.LIST(name expression)	Forms a variable-width node from all the nodes returned by the expression. The name gives the node name.
.SEXPN(expression)	FORMS a LISP S-expression from the nodes returned by the expression.
.EXECUTE	This treats the top of the node stack as a LISP expression and executes it.

Once again **- DON'T USE THESE CONSTRUCTS IN A DOMAIN!** They are for internal use only! The TACTICS interpreter is implemented using them.

2.4. Class of PARGEN Parsers

The PARGEN system produces parsers which scan left-to-right with explicit backup. The class of languages handled is less powerful than context-free. Some thought must be given to the ordering and content of the <parse-rule>s and <token-rule>s . Rules which could recur without scanning off a character are illegal. The worst case of this, left recursion, is of course illegal and must be removed by the author.

Backtracking rules must be included in the grammar whenever the complexity of the language to be recognized exceeds the power of an LL(1) parser. In particular a set of rules (a grammar) is LL(1) parseable if and only if whenever there is a rule of the form

```
ARULE = ALPHA / BETA ;
```

then the following conditions hold:

1. For no terminal symbol a do ALPHA and BETA derive strings beginning with a.
2. At most one of ALPHA and BETA can derive the empty string. The current implementation imposes the further constraint that only the last element of an alternation should directly derive the empty string with a .EMPTY.
3. If BETA can derive the empty string through a series of rule applications then ALPHA does not derive any strings beginning with a terminal symbol which is a member of the set of terminal symbols that can appear immediately to the right of ARULE in some sentential form.

Further information on LL(1) languages is found in [Aho79] which is the source of the explanation above.

As an example of fitting the rules into the constraints imposed by the parser generator the rule

```
RELOP = EXP "<" EXP / EXP "<=" EXP ;
```

would have to be changed to

```
RELOP = EXP ( "<=" / "<" ) EXP ;
```

or the less efficient

```
RELOP = EXP "<" EXP | EXP "<=" EXP ;
```

There are two reasons for this change. First, in the original RELOP the first nonterminal of the two alternatives was the same (EXP), so the first alternative would always be taken if an EXP object appeared in the input stream. In a sequence, if the first object is present in the input stream, then the rest of the sequence must be present or a syntax error is generated. Second, the "<=" must be tested for before the "<"; otherwise the "<" might match the first part of a "<=" in the input stream, and the wrong alternative would be taken.

A larger example of a complete external/internal domain language specification is given in Appendix APPX-PAREXAMP with some example programs in the language.

2.5. Using the Draco Parser Generator

For the purposes of this example we assume that the definition of a language is already prepared in Draco BNF and in the file LANG.DEF. The following is an example transcript.

```
DRACO>BUILD PARSER LANG LANG.DEF
{screen clears}
********... {one * signifys a line processed}
Parse Completed
{the extension defaults to .PAR}
LANG.PAR created
DRACO>
```

Once the parser has been created programs in the new language may be parsed by the Draco subsystem PARSE. If the parse had not successfully completed then the system would have stated why as in

```
******
ERR: Syntax error - rule EX3
   STMTS = STMT <scan>%<?:256>STMT  {a line from LANG.DEF}
{the <SCAN> marker is highlighted on some terminals and
 indicates the position in the input stream}
************** {more lines processed}
Parse Completed - 1 errors detected
```

The rule EX3 cited from PARGEN means the rule EX3 in the file PARGEN.DEF which describes the Draco BNF of parsers in Draco BNF. Other possible errors are discussed in the section on errors (see ERR:). PARGEN.DEF is reproduced in Appendix APPX-PARGEN and it gives the exact syntax of a domain language definition.

CHAPTER 3 BUILDING A PRETTYPRINTER WITH PPGEN

After the external and internal forms of a domain language have been described to Draco with BUILD, Draco must be told how the internal form is to printed out. The prettyprinter is also constructed using the BUILD subsystem which is described in this chapter. The prettyprinter for a domain language is used whenever Draco needs to communicate a program fragment to a user. In particular the transformation library constructing subsystem (BUILD), the transformation and refinement subsystem (TFMREF) and the program examination subsystem (EXAMINE) use the prettyprinter for a Domain Language.

The basic scheme for building prettyprinters is to describe a printing form for each node in the prefix internal form of the program. Carriage positioning may be added to these printing forms.

3.1. The Syntax of a Prettyprinter Description

In this section we use the standard BNF notation (see page) to describe the simple syntax of a prettyprinter definition.

```
<DracoPPdef>::= .PRETTYPRINTER <identifier> {<node-rule>} .END
<node-rule>::= <identifier> = <node-item> {<node-item>} ;
<node-item>::= <string> | <number> | # <number> | .LM |
               .COL( <number> ) | .SLM | .SLM( <number> ) |
               <list-scan> | .LM( <snumber> ) |
               .TREEPRINT
               ( <identifier> , <number> , <node-item> ) |
               .CHARTPRINT
               ( <identifier> , <number> , <node-item> , <node-item>
                 {<identifier> , <number> , <node-item> , <node-item>} ) |
               .CHARPRINT ( <number> ) |
               .LISTPRINT ( <node-item> )
<snumber>::= + <number> | - <number>
```

Where in the above BNF a <string> is a string of characters contained in double quotes ("). In this case <identifier> may be enclosed in angle brackets (<>) so that the identifier <FURD> is legal. The identifiers in angle brackets are used to define a print definition for the classes defined during transformation library construction (see Chapter 4). If an error occurs during the definition of a prettyprinter, then the rule cited is part of the file PPGEN.DEF which defines the syntax of a prettyprinter in Draco BNF. The current version of PPGEN.DEF is included in Appendix III of this document.

3.2. An Example of a Prettyprinter Description

Given the following parser definition from the previous chapter.

```
.DEFINE ASGNSTMT

ASGN = IDENTIFIER .LITERAL
       ":=" (EX1 / STRING .LITERAL)
       ";" .NODE(ASSIGN #2 #1) ;
EX1  = EX2 $("+" EX2 .NODE(ADD #2 #1)) ;
EX2  = EX3 $("*" EX3 .NODE(MPY #2 #1)) ;
EX3  = EX4 $("^" EX3 .NODE(EXP #2 #1)) ;
EX4  = IDENTIFIER .LITERAL
        ( "(" APARAMS ")" .NODE(FNCALL #2 #1) /
          .EMPTY ) /
       NUMBER .LITERAL /
```

```
              "(" EX1 ")" ;
     APARAMS = .TREE(APARAMS APSEQ EXP $("," EXP)) ;

     PREFIX : SPACES ;
     IDENTIFIER : SPACES .TOKEN ALPHA $<?:5>(ALPHA / DIGIT) .DELTOK
     NUMBER : SPACES .TOKEN DIGIT $DIGIT .DELTOK ;
     STRING : SPACES .TOKEN .ANY('") $.ANYBUT('") .ANY('") .DELTOK
     ALPHA : .ANY('A:'Z ! 'a:'z) ;
     DIGIT : .ANY('0:'9) ;
     SPACES : $.ANY(32) ;

     .END
```

The following is an example of a prettyprinter for the assignment statement parser given above.

```
     .PRETTYPRINTER ASGNSTMT

     ASSIGN  = #1 ":=" .LM #2 ";" ;
     ADD     = #1 "+" #2 ;
     MPY     = #1 "/" #2 ;
     EXP     = #1 "^" #2 ;
     FNCALL  = #1 "(" .LM #2 ")" ;
     APARAMS = #1 "," .SLM(60) #2 ;

     .END
```

In the example above ASGNSTMT is the name of the domain. Notice that there is one line for each possible prefix keyword in the internal form. The #1 in the lines refer to the next entry in the internal form node after the prefix keyword. The strings in quotes (") are strings which will be printed verbatim with no spaces on either side.

The .LM fixes the left margin to the position of the printing carriage at the time the .LM is encountered. This left margin prevails for the given rule and all rules called from it. A .LM with an unsigned number fixes the left margin at the column indicated by the number. A .LM with a signed number adds the given signed number to the left margin which preveiled when the prettyprint rule was entered and sets the left margin to the resultant value. The .SLM seeks the left margin set by the last .LM. If the carriage is before the current left margin, then .SLM will output tabs and spaces to get to the left margin. If the carriage is past the current left margin, then .SLM will perform a carriage return before tabbing and spacing over to the left margin. Remember if the output of the prettyprinter for a domain is to be acceptable input for the parser of the domain then that parser must accept tabs and spaces in positions where the prettyprinter indicates there will be whitespace in the prettyprinted output.

A numerical argument given to .SLM as in .SLM(65) will cause the .SLM to be effective only if the column position of the carriage is greater than the argument. Thus in the example above if a function call has more arguments than will fit on a line (60 columns), then the arguments which overflow will be printed under the first argument to the function.

If a number appears by itself in a prettyprinter description then it means to send that ASCII code to the output. It is alright to send literal carriage control characters to the output in that the prettyprinter understands where the carriage is but it is not a very good idea.

3.3. Output Device Dependent Codes

This is currently unimplemented but it is planned to provide an abstraction of different output devices for highlighting and cursor control. The basic information needed by the prettyprinter for these operations is the character sequence to send and the change in screen position the code causes. Some example control codes are given below:

```
   1. normal mode
   2. reverse video
   3. underlined
   4. blinking
   5. 16 combinations of the above
```

3.4 Prettyprinting TREEs and CHARTs

As mentioned in the previous chapter on parser construction, the parsers can build two kinds of special forms called TREEs and CHARTs. The prettyprinters have two special constructs to print these forms as TREEPRINT and CHARTPRINT.

The prettyprinting of a right-leaning tree is achieved using TREEPRINT. If we have the following TREE internal form:

```
.--------+--+------+
!        !  !      !
any-node a  .---+  b
            !   !
            IT  .-------+--+
                !       !  !
                IT-SEQ  ?  .-------+--+
                           !       !  !
                           IT-SEQ  ?  *OMEGA*
```

it could be prettyprinted in the IT prettyprint rule using

```
     .TREEPRINT(IT-SEQ,1,expression1,expression2)
```

The TREEPRINT construct would then look to see if the first subtree in the IT node is rooted with an IT-SEQ node. If so, the first subtree of that node would be prettyprinted followed by the evaluation of expression 1 and the prettyprinting of its second subtree. If the subtree is empty (i.e., it contains an *OMEGA*), then expression2 is evaluated. This recursive scheme is used to print trees of varying length. [NOTE 8] The use of TREEPRINT does not relieve the burden of producing prettyprinter rules for IT-SEQ type nodes. These prettyprinter rules are used only in priniting fragments for transformation and refinement pruposes. The CHARTs which can be created by the parsers are printed using CHARTPRINT. The use of CHARTPRINT is similar to the use of TREEPRINT. If we had the following node:

```
.---------+--+-----------------------+--------+
!         !  !                       !        !
any-node  ?  .--+                    .--+     ?
             !  !                    !  !
             A  .------+--+          B  .------+--+
                !      !  !             !      !  !
                A-SEQ  a  .------+--+   B-SEQ  1  .------+--+
                          !      !  !             !      !  !
                          A-SEQ  b  *OMEGA*       B-SEQ  2  *OMEGA*
```

we could use the following definition in the any-node prettyprinter rule to print a two column chart:

```
     .CHARTPRINT(A-SEQ,2,"->"," as ",B-SEQ,3,"[","]")
```

If used in the above chart internal form, it would result in the following output:

```
->a as [1]
->b as [2]
```

The CHARTPRINT directive pulls one element at a time from each of the right-leaning, internal-form trees and prettyprints them across the page. The two expressions associated with each tree are printed before and after the tree element. If all the trees of a chart are empty then the chartprint does no printing.

3.5 Special PrettyPrinter Functions

The CHARPRINT directive takes one subtree specification (e.g., #1) and attempts to treat the selected item as a number. If it is a number, it is sent as an ASCII code. For example,

```
       .CHARPRINT(2)
```

applied to the internal form

```
.---------+---+
!         !   !
any-node  23  65
```

would print an A since 65 is the ASCII code for an A. This is used where a domain language contains special quoted characters.

The .LISTPRINT prettyprinter directive is the printing analog of the .SEXPN and .LIST parsing directives. These are for internal use only and should not be used in domain languages.

3.6. Elements of a Prettyprinter Description

This section summarizes the elements which may appear in a prettyprinter line and what they do.

"abcd"	print a literal string
<number>	print the ASCII character
.LM	fix left margin at current position
.LM(<number>)	fix left margin at original margin plus the argument
.LM(-<number>)	fix left margin at original margin minus the argument
.LM(+<number>)	fix left margin at original margin plus the argument
.COL(<number>)	move to column number do crlf if necessary uses tabs and spaces
.SLM	seek left margin (from .LM) do crlf if necessary uses tabs and spaces
.SLM(<number>)	do .SLM if print column is greater than number
.TREEPRINT(<identifier>,<number>,<node-item>)	prints .TREE parsed internal forms
.CHARTPRINT(<identifier>,<number>,<node-item> {<identifier>,<number>,<node-item>})	prints .CHART parsed internal forms
.CHARPRINT(<number>)	prints explicit ASCII code
.LISTPRINT(<node-item>)	prints .LIST parsed internal forms

The printing of ASCII codes does not confuse the line column counter and these may be freely combined with the other directives. It is important to note that the prettyprinter will output tabs and spaces when it indents so that if the output from the prettyprinter is to be read back into PARSE then these characters must be acceptable input.

3.7. Producing a Successful Prettyprinter

Since the prettyprinter cannot print anything not in the internal form, a successful prettyprinter relies on an internal form which represents everything from in the external (syntactic) representation.

Our assignment statement example's internal form is deficient in that we cannot create a prettyprinter that can take an internal form and print it in a form which can be parsed into the original internal form. The problem with our example is that parentheses are not represented explicitly in the internal form. Implicitly the parentheses are represented by the hierarchy of the operators and the structure of the tree, but precedence information is available only to the parser, not the prettyprinter. As an example if we input

```
ANS := a*(B+C) ;
```

it would be prettyprinted by our defined prettyprinter as

```
ANS := a*B+C ;
```

While the internal representation is still correct the external representation is incorrect. Quite a bit of thought must go into what will be represented in the internal form and how this representation will be printed out so that it is pleasing to look at and accurate in its meaning. To solve our problem we should give the parentheses an internal representation by changing the line

```
"(" EX1 ")" ;
```

in the assignment statement example (see page) to the line

```
"(" EX1 ")" .NODE(PAREN #1) ;
```

Now to print this internal form node we must make the following prettyprinter definition.

```
PAREN = "(" #1 ")" ;
```

Now only the parentheses which appeared in the original program will be printed. It is useful to put a representation of practically every text level item into the internal form. This increases the accuracy of the prettyprinting, the range of possible transformations, and the range of possible refinements. It does, however, make the transformation definitions a bit more complex in that they will be responsible for removing and maintaining the text level forms.

3.8. Using the BUILD Subsystem to Create a Prettyprinter

This section presents an example transcript. Comments are in *{}* and user responses are underlined and terminated by a RETURN.

```
DRACO>BUILD LANG PRETTYPRINTER
{ file LANG.PPD contains prettyprinter definition }
{screen clears}
Prettyprinter Generator
******... {one * signifys a line processed}
Parse Completed 0 Errors Detected
NOTE: LANG.DPP created
{the extension defaults to .DPP}
DRACO>
```

As with parser generation, if prettyprinter generation had not completed an error giving a rule would have been printed out. In the case of prettyprinters this rule is contained in the file PPGEN.DEF which is reproduced as Appendix III of this manual. For other errors see Appendix X on errors. Appendix IV contains a complete example prettyprinter for the language presented in Appendix I.

CHAPTER 4 BUILDING A TRANSFORMATION LIBRARY WITH XFMGEN

After the Domain Analyst has settled upon an internal and external representation of the domain language and described this to Draco through PARGEN, the simple relationships between the objects and operations in his language must be described. These relationships are described as program transformations on the prefix internal structure of the domain.

The transformations will be used to simplify program fragments written in the language. These fragments may come from refinements of other domain languages into this language, the results of transformations on this language, or the use of PARSE to take in a program in this language.

The transformations usually represent relations which the Domain Analyst regards as being "obvious", such as x+0 implies [NOTE 5] x. The transformations will be used to strip the generality out of software components written in the language when they are used in a specific problem.

4.1. The Transformation Library and Metarules

The subsystem XFMGEN takes the view that transformations are *incrementally* added to a library of transformations for a domain. If a library does not exist then XFMGEN will create one.

There are two basic reasons for the incremental construction of the transformation library. First, it is hard to come up with all the useful transformations for a domain at once. Second, if automatic metarule generation (discussed below) is used it is computationally expensive to start a library from scratch.

4.1.1. Transformation Metarules

As XFMGEN reads in the transformations it has the capability to automatically produce "metarules". Briefly, these metarules give Draco information about what it can do next after it applies a transformation. The metarules state where it is important to apply which transformations in what order.

The metarules are expensive to produce since for every transformation added to the library, every transformation in the library (including the new one) must be examined for possible relations to the new transformation. The examination process itself is expensive. Thus, if we have a library of n-1 transformations the complexity of adding a new transformation is O(n) while the complexity of reconstructing the whole library is O(n*n). Since the library for a domain typically consists of 200-1000 transformations (sample size of two) the difference between incremental addition to a library and reconstructing the library is significant.

The specific scheme for automatically generating transformation metarules is described in [Neighbors80] Chapter 3 and Appendix II.

4.2. Specifying the Program Transformations

Unfortunately the program transformations must be specified in the *prefix internal form* of the domain language. The reason for this is that some transformations are not syntatically correct according to the external definition of the language. Also it is sometimes useful to insert special markers with a transformation (such as *EMPTY* or *UNDEFINED*) which will start off other transformations. As an example the transformation X/0=>*UNDEFINED* prevents undefined forms in a language from propagating. Remember if you insert such markers the domain prettyprinter must have a definition for printing them. Also all the markers which do not have an associated component must be removed before refinment is attempted or an error will occur.

4.2.1. The Syntax of a Transformation Insertion File

In this section we use the standard BNF notation (see BNF) to describe the syntax of a transformation insertion file which contains a packet of transformations to add to a library.

```
<DracoTIfile>::= {<TIcmnds>}
<TIcmnds>::= <pvardef> | <classdef> | <transdef> | (ERASEPVARS
<pvardef>::= (PVARS <identifier> {<identifier>} )
<classdef>::= (CLASS < <identifier> > <identifier> {<identifie
<transdef>::= (TRANS <identifier> <number> <lhs> <rhs> )
<rhs>::= <identifier> | <intform>
<lhs>::= <intform>
<intform>::= ( <identifier> {<rhs>} )
<identifier>::= <idchar> {<idchar>}
<idchar>::= A | ... | Z | a | ... | z |
            ! | # | % | & | *
```

A <number> is a simple sequence of numerals. Notice that the name of a class (the first identifier in the list) must be surrounded by angle brackets (<>) this helps to separate class names from a prefix keyword.

4.2.2. An Example of a Transformation Insertion File

If we refer back to our assignment statement example (see index) with the internal form descriptions, the following is an example transformation insertion file.

```
(PVARS X Y Z)
(CLASS <COMOP> ADD MPY)
(TRANS ADDX0 12 (ADD X 0) X)
(TRANS MPYX0 11 (MPY X 0) 0)
(TRANS EXPX0 11 (EXP X 0) 1)
(TRANS <COM>XY 5 (<COMOP> X Y) (<COMOP> Y X))
(TRANS PARENPAREN 12 (PAREN (PAREN X)) (PAREN X))
(TRANS LDISTMPYADD 5 (MPY X (PAREN (ADD Y Z)))
                     (PAREN (ADD (MPY X Y)(MPY X Z))))
(ERASEPVARS)
```

In the above example the nodes of the internal form are represented as lists of objects in parentheses. If a node contains a pointer to another node then that node is shown inside of the first node. Any identifier which is declared as a pattern variable (PVARS) will match a subtree or a constant. Thus the X,Y, and Z's in the example insertion file are all pattern variables.

A class (<COMOP> in the example above) represents a restricted pattern variable. Where the class name appears it will only match members of the class. The class <COMOP> represents the commutative operators and their commutativity is stated as transformation <COM>XY. A class can only contain identifiers but it can appear anywhere in an internal form. The transformations (TRANS) have a name, application code, left-hand-side (lhs), and right-hand-side (rhs). The lhs of a transformation must be an internal form (not a simple identifier) and the rhs of a transformation may be an internal form or an identifier. The application code states what kind of transformation this is and it is discussed in a following section. The lhs of a transformation is the form to be matched and the rhs of a transformation is the form which will replace it.

The ERASEPVARS command sets the list of current pattern variables to empty. The pattern variables are defined only for the transformations in the insertion file and they do not have to be the same ones used for transformations already in the library. The ERASEPVARS command is useful for concatenating together transformation insertion files to recreate a library from scratch.

Notice that the PAREN form which we added to the assignment statement example in the PARGEN section is cleverly used in the LDISTMPYADD transformation above. The lhs of the transformation assumes that there is a PAREN between the MPY and the ADD because in no other way could the tree have been constructed. The rhs is embedded in a PAREN because the precedence of ADD is less than MPY and the precedence must be maintained. The transformation PARENPAREN (and some others) is needed to maintain only the PAREN's needed. To reiterate, the selection of what to represent in the internal form and how to represent it is a difficult process.

4.2.3. Elements of a Transformation Insertion File

This section summarizes the commands to XFMGEN which may appear in a transformation insertion file.

Syntax	**Meaning**
(CLASS <classname> <identifier>...<identifier>)	This declares a restricted pattern variable named <classname> which can match any of the identifiers given. The <classname> may appear anywhere in a transformation.
(PVARS <identifier>...<identifier>)	This declares all the identifiers as unrestricted pattern variables. If a pattern variable appears twice in a pattern then the objects it matches must be the same.
(TRANS <transname> <application-code> <lhs> <rhs>)	This describes a transformation with name <transname> and other fields as shown.
(ERASEPVARS)	This erases all current pattern variables but not classes.

4.2.4. The Application Code of a Transformation

In the transformations above and in Appendix V, the application codes follow the following rough guidelines [NOTE 6]. EC stands for enabling conditions.

100-up	Markov algorithm (can enlarge locale
95-99	Always do this transformation (no EC's)
90-94	Always try this transformation (has EC's)
85-99	Convert to canonical form (no EC's)
80-84	Convert to canonical form (has EC's)
75-79	Reverse canonical form (no EC's)
70-74	Reverse canonical form (has EC's)
60-69	Operator arrangement (no EC's)
50-59	Operator arrangement (has EC's)
40-49	Flow statement arrangement (no EC's)
30-39	Flow statement arrangement (has EC's)
20-29	Program segment arrangement (no EC's)
10-19	Program segment arrangement (has EC's)
00-09	Start a Markov algorithm

These are completely arbitrary and you may make up your own codes. The codes for source-to-source transformations run from 1 to 99 while Markov algorithms use above 100 and 0. The Draco system knows nothing about particular application codes except that odd codes represent transformations with enabling conditions.

Since the current system does not support the checking of enabling conditions it stops and asks the user before it applies any transformation with an odd code. The codes are used in the TFMREF subsystem by the TRANSFORM command.

In the future the application codes may be used for a best first style lookahead so the better transformations should have higher application codes.

4.3. The Catalog of Transformations for a Domain

When XFMGEN produces a new library it gives the option for a catalog listing. The catalog is a listing of all the transformations in alphabetical order prettyprinted by the domain prettyprinter. To prettyprint classes they must be defined to the prettyprinter. Our example above would require the line

```
                    <COMOP> = #1 "<COMOP>" #2 ;
```

to be added to the prettyprinter for the assignment statement example in the section on PPGEN. For our example above the catalog would have looked like:

```
4/6/82   18:00:00   LANG.TLB
<COMOP> = {ADD,MPY}
<COM>XY: 5  ?X<COMOP>?Y  =>  ?Y<COMOP>?X
```

```
     ADDX0: 12   ?X+0  => ?X
     EXPX0: 11   ?X^0  => 1
     LDISTMPYADD: 5   ?X*(?Y+?Z)  =>  (?X*?Y+?X*?Z)
     MPYX0: 5   ?X*0  =>  0
     PARENPAREN: 12   ((?X))  => (?X)
```

The first line gives the date, time and the name of the file which contains the library. The pattern variables in each transformation are preceeded by a question mark (?). These catalog listings are useful references when working with the TFMREF subsystem. Appendix V presents an example catalog of transformations for the language defined in Appendix I. It is interesting to note how the special marker *UNDEFINED* is used in Appendix V and how the metarules can take advantage of such markers.

4.4. Using the Transformation Builder System: XFMGEN

In this example we assume that the file PARSER.TFM (see figure below) contains some transformations to be inserted into the library of the PARSER domain. We further assume that the PARSER domain currently has no transformation library.

```
(PVARS x y z)
(TRANS true 95 (compr x x) *true*)
(TRANS false 97 (uneq x x) *false*)
(TRANS not# 95 (negation (uneq x y)) (compr x y))
(TRANS not= 95 (negation (compr x y)) (uneq x y))
(TRANS ifelim1 99 (ifthen *true* x) x)
(TRANS ifelim2 99 (ifthen *false* x) *empty*)
(TRANS ifelim3 99 (ifelse *true* x y) x)
(TRANS ifelim4 99 (ifelse *false* x y) y)
(ERASEPVARS)
```

We activate the XFMGEN subsystem through the BUILD command. In the first session below, XFMGEN notes it is creating a new transformation library. It follows a list of the transformations inserted into the library, anf the user is offered a prettyprinted version of the transformation library. Finally, the transformation library PARSER.TLB is created.

```
    DRACO>build (DOMAIN NAME) parser (DOMAIN PART) transformation-library
    Transformation Library Builder working on PARSER domain
    NOTE: creating a new transformation library
    true false not# not= ifelim1 ifelim2 ifelim3 ifelim4
    Prettyprinted Transformation Catalog Listing? (Y/N)> N
    NOTE: PARSER.TLB created
    DRACO>
```

When new transformations have to be added to the library, the procedure to follow is similar. Let us assuem that as some point in the future the insertion file PARSER.TFM includes the following transformations:

```
(PVARS x y z)
(TRANS parenthelim 99 (paren (paren x)) (paren x))
(TRANS parenthelim2 98 (stmnt (paren x) stmnt) (stmnt x stmnt))
(TRANS parenthelim3 98 (stmnt (paren x) *OMEGA*) (stmnt x *OMEGA*))
(TRANS stmntelim1 12 (stmnt *empty* stmnt) stmnt)
(TRANS stmntelim2 12 (stmnt *empty* *OMEGA*) *OMEGA*)
(TRANS bodyelim 12 (body (stmnt x *OMEGA*)) x)
(TRANS bodyelim2 12 (body (stmnt (*empty* *OMEGA*)) *empty*)
(ERASEPVARS)
```

The following dialogue updates PARSER.TLB with the content of the new transformation insertion file. In this second session, XFMGEN, the Transformation Library Builder, notes that a transformation library (PARSER.TLB) already exists.

```
DRACO>build (DOMAIN NAME) parser (DOMAIN PART) transformation-library
Transformation Library Builder working on PARSER domain
NOTE: adding to an exisiting transformation library
parenthelim parenthelim2 parenthelim3 stmntelim1 stmntelim2
bodyelim bodyelim2
Prettyprinted Transformation Catalog Listing (Y/N)> Y
NOTE: PARSER.CAT created
NOTE: PARSER.TLB created.
```

A prettyprinted version of the updated library (PARSER.CAT) is shown below.

```
3/24/83   1:49:2pm   PARSER.TLB
bodyelim: 12  ?x  => ?x
```

```
bodyelim2: 12  *empty*  =>  *empty*
false: 97  ?x # ?x  => *false*
ifelim1: 99  if *true*
     then
        ?x
     end if ;  =>  ?x
ifelim2: 99  if *false*
     then
        ?x
     end if ;  =>  *empty*
ifelim3: 99  if *true*
     then
        ?x
     else
        ?y
     end if ;  => ?x
ifelim4: 99  if *false*
     then
        ?x
     else
        ?y
     end if ;  =>  ?y
not#: 95  (?x # ?y) not  =>  ?x = ?y
not=: 95  (?x = ?y) not  =>  ?x # ?y
parenthelim: 99  ((?x))  =>  (?x)
parenthelim2: 98  (?x);
            =>  ?x;
parenthelim3: 98  (?x);
            =>  ?x;
stmntelim1: 12  *empty*;
          => stmnt
stmntelim2: 12  *empty*;
          =>
true: 95  ?x = ?x  =>  *true*
```

Usually as the small packets of transformations in insertion files are added to the transformation library, the user should concatenate these packets together into a file in case the transformation library ever need be generated from scratch. The following file combines the transformations from this section.

```
(PVARS x y z)
(TRANS true 95 (compr x x) *true*)
(TRANS false 97 (uneq x x) *false*)
(TRANS not# 95 (negation (uneq x y)) (compr x y))
(TRANS not= 95 (negation (compr x y)) (uneq x y))
(TRANS ifelim1 99 (ifthen *true* x) x)
(TRANS ifelim2 99 (ifthen *false* x) *empty*)
(TRANS ifelim3 99 (ifelse *true* x y) x)
(TRANS ifelim4 99 (ifelse *false* x y) y)
(ERASEPVARS)
(PVARS x y z)
(TRANS parenthelim 99 (paren (paren x)) (paren x))
(TRANS parenthelim2 98 (stmnt (paren x) stmnt) (stmnt x stmnt))
(TRANS parenthelim3 98 (stmnt (paren x) *OMEGA*) (stmnt x *OMEGA*))
(TRANS stmntelim1 12 (stmnt *empty* stmnt) stmnt)
(TRANS stmntelim2 12 (stmnt *empty* *OMEGA*) *OMEGA*)
(TRANS bodyelim 12 (body (stmnt x *OMEGA*)) x)
(TRANS bodyelim2 12 (body (stmnt (*empty* *OMEGA*)) *empty*)
(ERASEPVARS)
```

CHAPTER 5
BUILDING A COMPONENT LIBRARY WITH REFGEN

5.1 The Constituent Parts of a Component

An example component for exponentiation is shown below. The component provides the semantics for EXP internal form nodes for the language SIMAL which is <Strong>not</Strong> a domain-specific language, but will be used in examples so that the reader will not have to learn a domain-specific language at this point.

```
COMPONENT: EXP(A,B)
PURPOSE: exponentiation, raise A to the Bth power
```

```
IOSPEC: A a number, B a number / a number
DECISION:The binary shift method is O(ln2(B)) while
       the Taylor expansion is an adjustable number
       of terms. Note the different conditions for
       each method.
REFINEMENT: binary shift method
CONDITIONS: B an integer greater than 0
BACKGROUND: see Knuth's Art of ... Vol. 2,
    pg. 399, Algorithm A
INSTANTIATION: FUNCTION,INLINE
RESOURCES: none
CODE: SIMAL.BLOCK
  [[ POWER:=B ; NUMBER:=A ; ANSWER:=1 ;
     WHILE POWER>0 DO
       [[ IF POWER.AND.1 # 0
     THEN ANSWER:=ANSWER*NUMBER ;
  POWER:=POWER//2 ;
  NUMBER:=NUMBER*NUMBER ]] ;
     RETURN ANSWER ]]
END REFINEMENT
REFINEMENT: Taylor expansion
CONDITIONS: A greater than 0
BACKGROUND: see VNR Math Encyclopedia, pg. 490
INSTANTIATION: FUNCTION,INLINE
ASSERTIONS: none
ADJUSTMENTS: TERMS[20] - number of terms,
error is approximately (B*ln(A))^TERMS/TERMS!
CODE: SIMAL.BLOCK
  [[ SUM:=1 ; TOP:=B*LN(A) ; TERM:=1 ;
     FOR I:=1 TO TERMS DO
       [[ TERM:=(TOP/I)*TERM ;
  SUM:=SUM+TERM ]] ;
     RETURN SUM ]]
END REFINEMENT
END COMPONENT
```

Each component has a name and a list of possible arguments in the COMPONENT field.

The name is the prefix keyword of the internal form nodes to which the component applies. The list of possible arguments name the subtrees of the internal form node. If a node has a variable number of subtrees, a name prefaced by a ">" is used to denote the rest of the subtrees in the node.

A prose description of what the component does is given by the PURPOSE field. If the component takes objects as arguments and/or produces objects, then the type of these objects in terms of the objects in the domain is given in the IOSPEC field of the component. The DECISION field presents a prose description of the possible refinements of the component and the considerations involved in choosing between the alternatives.

Finally, there is a set of refinements of the component which represent a possible implementation of the component in terms of the objects and operations of other domains.

The first REFINEMENT in the set of refinements is the default refinement. In the absence of any other information, Draco will attempt to use this refinement first. Each REFINEMENT has a name and a BACKGROUND which is a prose description of the method the refinement implements and reference to where more information about the method may be found.

The CONDITIONS field of a refinement lists conditions which must be true before the component may be used. There are basically two kinds of conditions: conditions on the domain objects on which the component operates and conditions on previously made implementation decisions. The conditions on the domain objects are local to the locale where the component will be used. The conditions on the implementation decisions are global to the domain instance being refined. The ASSERTIONS field of a refinement makes assertions about the implementation decisions the component makes if it is used. The assertions are the opposites of the conditions on implementation decisions. The management of assertions and conditions is discussed in more detail in [Neighbors80].

The RESOURCES field of a refinement states what other components will be required to perform initialization if the refinement is chosen. The resource components are program parts which are executed before the resulting program begins execution (initialization phase) and they create information resources for the refinements used in the program.

An example use of a resource is a refinement for cosine which interpolates a table of cosines during execution. The table must be built during the initialization phase and the name of the table must be passed to the interpolation refinement of the component cosine. This is achieved by building a refinement which interpolates tables and requires a resource component which builds interpolation tables.

The ADJUSTMENTS field of a refinement states fine tuning settings for a refinement, the meaning of the adjustment, and a default setting. An example adjustment term might adjust the accuracy of a refinement or limit the amount of time spent in calculating in the refinement.

The GLOBAL field lists all names used in the refinement which are not to be renamed. The primary use of a GLOBAL definition is to define variable names which are reserved by a domain and cannot be renamed. The SNOBOL variable &ANCHOR is an example global. GLOBAL definitions should seldom be used and are always suspect. They seem to stem from a poor analysis of a domain. Labels which are defined in the refinement are defined in the LABELS field of the refinement.

The way a refinement may be inserted into the internal form tree during refinement is governed by the INSTANTIATION field of the refinement. The three modes of instantiation are INLINE, FUNCTION, and PARTIAL. More than one instantiation may be given for a refinement with the first one listed being the default instantiation. INLINE instantiation means the refinement is substituted directly into the internal form tree with all variables used in the refinement renamed (including labels) except for the arguments and those declared global. FUNCTION instantiation substitutes a call for the component in the internal form tree and defines a function using the refinement for the body. A new function is defined only if the same function from the same domain has not already been defined. PARTIAL instantiation substitutes a call for the component in the internal form tree with some of the arguments already evaluated in the body of the function defined. Limitations are placed on the partially evaluated forms allowed. When a function is defined the defining domain, component name, and a version number are used to differentiate between functions of the same name in different domains and FUNCTION and PARTIAL versions of the same function in the same domain.

The final field of a refinement is either a DIRECTIVE to Draco or the internal form of a domain. The internal form of a domain may be described either in a parenthesized tree notation with the INTERNAL:domain directive or it may be specified in the external form (domain language) of the domain with the CODE:domain.nonterminal directive. The CODE directive causes the parser for the specified domain to be read in and started to recognize the given nonterminal symbol. A DIRECTIVE to Draco is one of the following alternatives: view the component as a function definition by the user program, view the component as a function call, defer from refining this component, and remove the node which invoked this component from the internal form tree. The Draco DIRECTIVEs are used when a domain language is defined which allows function definitions, functions calls, and such things as refinements for comments which remove them from the program since they are saved in the refinement history.

Not all the component and refinement fields are required for each component definition. Basically the only required fields are COMPONENT, REFINEMENT, INSTANTIATION and CODE.

5.2. The Motivation for Libraries of Components

Components are placed into libraries in much the same way and for much the same reason that transformations are placed into libraries. The processing of a single component for inclusion in the component library of a domain is very expensive. For each refinement in the component, the parser for the domain(s) in which the refinement is written must be loaded to parse the external form into internal form. Once the code for the refinement is in internal form, the agendas of the internal form are annotated with transformations of interest from the transformation library of the target domain. These transformation suggestions are made in much the same way that transformation suggestions are made when a domain language program is parsed as discussed on <A HREF="#TRANSSUGGEST">page</A>. The transformation suggestions will point out things of interest when the refinement is used. Thus, Draco supports a component library construction facility where a group of components may be replaced or added without disturbing the other components in the library.

5.3. Building a Component Library

The REFGEN subsystem in DRACO supports the construction of libraries of components. The process is activated by using a variant of the BUILD command. The components to be inserted in the library are kept in a <domain-name>.REF file. When the building process begins, the Refinement Library (<domain-name>.RLB file) may not exist, and it is created from scratch. If the library is not empty, the components in the <domain-name>.REF file are incorporated into it; those that were already there are updated according to the new definition from the <domain-name>.REF file. Thus, this variant of the BUILD command is used both for creating and updating the refinement libraries.

To illustrate the dialogue with REFGEN we will use a set of refinement components of DRACO itself as input: DRACO.REF. As we already have a Refinement Library for Draco, REFGEN NOTEs it and provides a list of the components defined. At this point, if no <domain-name>.REF or <domain-name>.DEC files are found in the user directory, REFGEN will flag an error, and the process will be aborted. REFGEN prints asterisks as it parses each line of the component definitions.

```
DRACO>build (DOMAIN NAME) draco (DOMAIN PART) component-library
```

```
Component Library Builder working on DRACO domain
NOTE: existing component library contains:
*APARAMS* *APARAMS-SEQ* *FPARAMS* *FPARAMS-SEQ* *LABELS*
*LABELS-SEQ* *LOCALS* *LOCALS-SEQ* *PARTIALS* *PARTIALS-SEQ*
*PROCCALL* *PROCDEF* *PROCLIST* pROCLIST-SEQ* *SEQUENCE*

NOTE: insertion file components replace library components

Component *PROCLIST*            **
    Refinement LISP function list as a read execution sequence
                        *******

Component *PROCLIST-SEQ*        **
    Refinement LISP function list as a read execution sequence
                        *******

Component *PROCDEF*             **
    Refinement LISP function definition
                        ********

etc ...
```

The following figure is a fragment of the input file, DRACO.REF, showing some of the components that were processed through the dialog transcribed above.

```
COMPONENT:*PROCLIST*(PROCS)
PURPOSE: The list of functions known to Draco
REFINEMENT: LISP function list as a read execution sequence
INSTANTIATION: INLINE
INTERNAL:LISP
(LISPPGM {{PROCS}})
END REFINEMENT
END COMPONENT

COMPONENT:*PROCLIST-SEQ*(PROCS1,PROCS2)
PURPOSE: The list of functions known to Draco
REFINEMENT: lisp FUNCTION LIST AS A READ EXECUTION SEQUENCE
INSTANTIATION: INLINE
INTERNAL:LISP
(LISPSEQ  {{PROCS1}} {{PROCS2}})
END REFINEMENT
END COMPONENT

COMPONENT *PROCDEF*(DOMAIN,NAME,VERSION,BODY,
            FPARAMS,LOCALS,PARTIALS,LABELS)
PRUPOSE: A Draco function definition.
REFINEMENT: Lisp function definition.
BACKGROUND: WARNING: could cause naming problems!
INSTANTIATION: INLINE
CODE: LISP.SEXPN
{{ (DE {{NAME}} {{FPARAMS}} (PROG {{LOCALS}}
        (RETURN {{BODY}}))) }}
END REFINEMENT
END COMPONENT
```

CHAPTER 6
CONVERTING A PROGRAM TO INTERNAL FORM WITH PARSE

When a new system needs to be built which can be described in one of Draco's domain languages needs to be built,the PARSE subsystem is used to convert the domain language program into the internal form Draco can manipulate. The PARSE subsystem reads in the parser built by the PARGEN subsystem for the domain language.

If there are transformations defined for the domain, they are used to automatically suggest possible transformations for the problem at hand. These suggestions are attached to the programs internal form and are used when the program is transformed and refined by the TFMREF subsystem. If no transformation library exists for the domain the message "ERR: transformation library <DOMAIN>.TLB unavailbale for suggestions" will be issued. Even with this message, an internal form will be created for the program; but this program will not have suggestions for transformations.

6.1. Using the PARSE Subsystem

The following is an example interaction with the PARSE subsystem. LANG is the name of the domain and PROG.PGM is a file which contains a program written in LANG.

```
DRACO>**PARSE (DOMAIN NAME) LANG (SOURCE FILE) LANG.PGM**
{Draco loads the parser for the domain}
Parsing LANG from file LANG.PGM
*******... {one * signifys a line read}
Parse Completed 0 errors detected
{the extension defaults to .INT}
PROG.INT created
DRACO>
```

The error conditions and error messages are similar to those for PARGEN and PPGEN in that if an error occured during parsing a rule name and offending line would be printed. In this case the offending line would come from PROG.PGM and the rule would be contained in the parser definition for the domain, LANG.DEF.

6.2. How Transformations are Suggested in PARSE

In the example dialogue above the question "Automatic Suggestion of Transformations ?" was asked. If the response to this question was yes (Y) then in the internal form that PARSE builds some suggestions of transformations are made. The transformations are suggested by the prefix keyword at each node in the tree and no further matching is done. Thus, any transformation which *could* possibly apply is suggested. Many of those suggested will not apply.

The suggestion mechanism assumes that the TRANSFORM command of the TFMREF subsystem will be used to weed out any inapplicable transformations very quickly. If you plan to apply each transformation by hand individually in the TFMREF subsystem then the automatic suggestion of transformations option should not be taken. If the option is taken then the suggested transformations are ordered by their application codes (see XFMREF).

6.3. Multiple Domains

When writing programs in one domain we can use statements in other domains. To do this we need to signal to the parser that a change of domains will take place. This is done with the following construct:

```
{{<DOMAIN-NAME>.<RULE-NAME>{{<statements>}}}}
```

No blanks are allowed between the braces and text.

The rule name can be the main rule of the parser of the inside domain, or it can be a specific rule just related to the statement(s) we want to use.

CHAPTER 7
USING TFMREF THE PROGRAM MANIPULATION SUBSYSTEM

The transformation and refinement subsystem of Draco (TFMREF) is used by a System Specialist to refine a program written in a domain language into an executable language. Before the TFMREF system can manipulate a program it must be converted into the prefix internal form for the domain by PARSE.

The basic cycle of a System Specialist using TFMREF is to transform the program to remove inefficiencies and then to refine the program by selecting an appropriate refinement (software component) to implement the primitives of the domain used in the program. This cycle is repeated over and over with the software components introducing meaning and the transformations stripping the generality out of the software components.

Each time a refinement is made a record is kept so that the EXAMINE subsystem can account for the purpose of any line at any level of refinement in the resulting system.

7.1. The TFMREF Commands Which Set the Context

When working on the refinement of a large and complex system in an iterim stage ofdevelopment, it is important to bound the context of refinement considered by the System Specialist. In TFMREF this is acheved with three mechanisms DOMAIN, INSTANCE, and LOCALE.

While a program is being refined it may exist as program fragments in many domains at once. The different domains are being used as modelling domains for the problem. So the first index of bounded context is which DOMAIN the System Specialist intends to work in.

Once the domain is selected then which INSTANCE of this domain is desired provides a second restriction of context.

Finally, provision is made for the System Specialist to specify a restricted LOCALE within the instance of the selected domain.

All of these narrowings of the context serve to focus both the System Specialist and TFMREF in examining what can be done. Initially when TFMREF is entered it requests the file containing the internal form of the program to be refined, the domain, and the instance. The following sections describe the commands which restrict the context. The selected context can be displayed in a shorthand notation using the INFO command.

7.1.1. The DOMAIN command

DOMAIN <domain name>

The domain command is used to change the domain the System Specialist whishes to work in. TFMREF automatically performs a DOMAIN command upon entry. All parts (prettyprinter, transformations, and refinements) of the old domain are removed from memory and the prettyprinter (if one exists) for the new domain is loaded. The instance is unselected when a old domain is removed.

7.1.2. The INSTANCE command

INSTANCE

The INSTANCE command is used to change the instance of the currently selected domain to some other instance of the domain in the selected program. To select the instance TFMREF scans the program selected looking for program fragments written in the selected domain. If it finds one it prettyprints it out to a certain depth using "..." to indicate supressed detail. It then asks the user if this was the instance he had in mind. If so it selects the instance. If no instance is selected then the other commands in TFMREF which require an instance will either automatically select an instance or not function until an instance is selected.

7.1.3. The LOCALE Command

LOCALE [No. of levels]

The LOCALE command restricts the context to a part of the selected instance by traversing the prefix tree internal form. If you expect to apply transformations one at a time then the locale command must be used to set the location of application. This is a tedious operation since TFMREF must be able to traverse the internal form of any domain without the System Specialist having to know the internal form of the domain.

When the locale command is given the system prints the currently selected locale and requests a number. A negative number (-n) moves up the tree n levels (limiting to the instance root) while a positive number (n) moves down the nth subtree. Error messages are printed when the number of levels exceed the number of available subtrees.

The PP command prettyprints the current locale if one has been selected.

7.2. The Miscellaneous TFMREF Commands

This section presents the commands of the TFMREF subsystem which either present or save environmental information. These commands are not specific to locale, transformations, or refinements.

7.2.1. The PP Command

PP

The PP command prints the selected locale completely without the "..." shorthand. The output may be safely aborted with a control-O.

7.2.2. The INFO Command

INFO

The INFO command prints the time, date, program file you are working on, the domain selected, what is in memory (PP=prettyprinter, TFM=transformations, REF=refinement index), the short version of the instance, and the short version of the locale.

7.2.3. The HARDCOPY Command

HARDCOPY <filename.ext>

The HARDCOPY command enables the System Specialist to get a disk file of the prettyprinted version of the entire domain instance. When a program has been refined from one domain language into an executable language this command must be used to get a copy of the resulting program.

7.2.4. The SAVE Command

SAVE

The SAVE command saves the entire prefix internal form, suggested transformations and refinement record, over the old program file. The name of the program file is given by the INFO command. Upon EXITing the TFMREF subsystem the user is automatically asked if he wants to save the internal form. This command is included for incremental saves to avoid the problem of a crash destroying the entire sessions work.

7.2.5. The EXIT Command

EXIT

The EXIT command exits the TFMREF subsystem, asks if a SAVE should be done, and then returns to Draco.

7.3. A Summary of the TFMREF Commands

The TFMREF Subsystem has a HELP command which prints out the following summary of the TFMREF commands.

```
The TFMREF commands are:
 DOMAIN    - specify a new domain to work with
 INSTANCE  - specify which instance of the chosen domain to work with
 NOINSTANCE- remove any instance selection for autoselection
 LOCALE    - specify a subpart of the instance to work with
 PP        - display the locale selected
 INFO      - print out environment stats
 UNLOAD-TRANSFORM - unload the transformations for the domain
 UNLOAD-REFINE    - unload the components for the domain
 REFLRU    - set the LRU stack length for no. of components in mem.
 ?TRANS    - print out a transformation
 ?CLASS    - print out a class
 SUGGEST   - suggest transformations to apply to the locale
 APPLY     - apply a transformation to the locale
 TRANSFORM - scan for transformations in the current locale
 ANNOTATE  - attach transformations in <domain>.TLB to internal form
 REFINE    - scan for refinements in the current locale
 TACTICS   - envoke the tactics subsystem

 SAVE      - save all the work so far
 HARDCOPY  - prettyprint the instance to a file
 EXIT      - exit the TFMREF subsystem
 HELP      - this list
```

The TFMREF commands for transformation and refinement are described in Chapter 8 and Chapter 9 respectively. Tactics are described in Chapter 10.

7.4. An Example Session with TFMREF

This section presents a session with TFMREF. In the session don't be concerned with the meaning of the domain language being manipulated (SIMAL see Appendix I). What is of concern in the example is the way TFMREF commands interact with the user to manipulate this small, trivial example. In the transcript the *{}* denote comments and underlining denotes user responses. All user responses are terminated with a RETURN.

```
DRACO>TFMREF (PROBLEM FILE) quad
{extension defaults to .INT}
{screen clears}
Transformation and Refinement Subsystem
NOTE: file last modified on 5/25/79 22:32:49
The modules are:
        DRACO.START.0()[]{}
                <DL:DOC_DOCUMENT>
        (Y/N)>n
TFMREF>Domain (DOMAIN NAME) SIMAL
{ALGOL-like lang for examples}
NOTE: DRACO domain being removed
TFMREF>instance
    .PROGRAM QUADRATIC              {shorthand printout}
    $QUADRATIC
    [[LOCAL ...;
      ...;
      ...]]
      $
    .END
                _
                (Y/N)>Y                 {is this the instance?}

TFMREF>PP                  {lets see all of program}
    .PROGRAM QUADRATIC
    $QUADRATIC
    [[LOCAL A,B,C,ROOT1,ROOT2;
      LOOP:
        PRINT("QUADRATIC EQUATION SOLVER",CRLF);
      PRINT("INPUT A,B,C PARAMETERS ");
      A:=READNUM;
      IF A=0 THEN RETURN ;
      B:=READNUM;
      C:=READNUM;
      ROOT1:=(-B+SQRT(B^2-4*A*C))/(2*A);
      ROOT2:=(-B-SQRT(B^2-4*A*C))/(2*A);
      PRINT("THE ROOTS ARE: ",ROOT1," AND ",ROOT2,CRLF);
      GOTO LOOP]]
      $
    .END

TFMREF>LOCALE               {lets restrict the context}
    .PROGRAM QUADRATIC
    $QUADRATIC
    [[...
      ...]]
      $
    .END

TFMREF>locale 1
    $QUADRATIC
    [[LOCAL ...;
      ...;
      ...]]
      $

TFMREF>locale 1
    [[LOCAL A,B,C,ROOT1,ROOT2;
      LOOP:
        ...;
      ...;
      ...]]

TFMREF>locale 2
    LOOP:
      PRINT(...);
    PRINT(...);
    ...;
    ...

TFMREF>locale 2
    PRINT("INPUT A,B,C PARAMETERS ");
    A:=READNUM;
    ...;
    ...

TFMREF>locale 2
    A:=READNUM;
    IF ... THEN ...;
    ...;
    ...

TFMREF>locale 2
    IF A=0 THEN RETURN ;
    B:=READNUM;
    ...;
    ...

TFMREF>locale 2                {progressively deeper into program}
            B:=READNUM;
    C:=READNUM;
    ...;
    ...

TFMREF>locale 2
            C:=READNUM;
    ROOT1:=...;
    ...;
    ...

TFMREF>locale 2
            ROOT1:=.../...;
    ROOT2:=...;
    ...;
    ...

TFMREF>locale 1
            ROOT1:=(...)/(...)

TFMREF>locale 2
    number too large                       {an error - try again}

TFMREF>locale 1
    (...+...)/(2*A)

TFMREF>locale 1
    (-B+SQRT(...))

TFMREF>locale 1
    -B+SQRT(...)

TFMREF>locale 2
    SQRT(...-...)

TFMREF>locale 2
    number too large

TFMREF>locale 1
    B^2-...*C

TFMREF>locale 1
    B^2-4*A*C

TFMREF>locale 1
    B^2
 {ok lets look at the ^ operator}

TFMREF>PP
    B^2                        {yes were really there}
```

```
TFMREF>**SUGGEST**                  {ask for transformation suggestions}
    10 OIDDEF                   {transformation name and application code}
    7 EXPX2                     {in application code order}
    3 <OP>XIF
    2 <OP>IFX

TFMREF>?TRANS EXPX2             {what does expx2 transformation do}
    transformations loaded      {first get the transformation library}
    EXPX2: 7  ?X^2  =>  ?X*?X   {it makes ^ into *}

TFMREF>?TRANS OIDDEF            {what is an OIDDEF}
    OIDDEF: 10  ?X<OIDOPS>1  => ?X  {1 identity operators}

TFMREF>?CLASS <OIDOPS>          {which operators are 1 identity}
    <OIDOPS> = {MPY,EXP}        {the set MPY and EXP}

TFMREF>APPLY EXPX2               {lets do expx2 here at B^2}
    EXPX2: 7  B^2  =>  B*B    (Y/N)>Y  {before and after}

TFMREF>LOCALE 1                  {zoom out from where tfm took place}
    B*B

TFMREF>locale -1
    B*B-4*A*C

TFMREF>locale -2
    SQRT(...-...)

TFMREF>locale -100               {get me to the instance root now}
    .PROGRAM QUADRATIC
    $QUADRATIC
    [[...
      ...]]
      $
    .END

TFMREF>PP                        {did the program change?}
    .PROGRAM QUADRATIC
    $QUADRATIC
    [[LOCAL A,B,C,ROOT1,ROOT2;
      LOOP:
        PRINT("QUADRATIC EQUATION SOLVER",CRLF);
      PRINT("INPUT A,B,C PARAMETERS ");
      A:=READNUM;
      IF A=0 THEN RETURN ;
      B:=READNUM;
      C:=READNUM;
      ROOT1:=(-B+SQRT(B*B-4*A*C))/(2*A);     {we changed this line}
      ROOT2:=(-B-SQRT(B^2-4*A*C))/(2*A);
      PRINT("THE ROOTS ARE: ",ROOT1," AND ",ROOT2,CRLF);
      GOTO LOOP]]
      $
    .END

TFMREF>TRANSFORM (LO CODE) 3 (HI CODE) 12 (APPROVAL MODE) ask
{The easier way to do transformations. TRANSFORM applies
 transformations throughout the currently selected locale.}
{3 and 12 mean application codes 3 to 12 and ask me before
 doing any transformations. All transformations with enabling
 conditions will always ask anyway.}
    EXPX2: 7  B^2  =>  B*B    (Y/N)>Y

TFMREF>PP                        {lets see where the change was made}
    .PROGRAM QUADRATIC
    $QUADRATIC
    [[LOCAL A,B,C,ROOT1,ROOT2;
      LOOP:
        PRINT("QUADRATIC EQUATION SOLVER",CRLF);
      PRINT("INPUT A,B,C PARAMETERS ");
      A:=READNUM;
      IF A=0 THEN RETURN ;
      B:=READNUM;
      C:=READNUM;
      ROOT1:=(-B+SQRT(B*B-4*A*C))/(2*A);
```

```
       ROOT2:=(-B-SQRT(B*B-4*A*C))/(2*A);    {this line changed}
       PRINT("THE ROOTS ARE: ",ROOT1," AND ",ROOT2,CRLF);
       GOTO LOOP]]
       $
     .END

 TFMREF>INFO                       {what am I doing with what when}
     5/25/79  23:28:41
     PROGRAM: QUAD.INT
     DOMAIN: SIMAL  PP TFM         {prettyprinter and transforms in memory}
     INSTANCE: .PROGRAM QUADRATIC  {instance and locale are same}
               $QUADRATIC
               ...
                 $
               .END

     LOCALE: .PROGRAM QUADRATIC
             $QUADRATIC
             [[...
               ...]]
               $
             .END

 TFMREF>unload-transform           {remove transformations}

 TFMREF>locale
     .PROGRAM QUADRATIC
     $QUADRATIC
     [[...
       ...]]
       $
     .END

 TFMREF>locale 1
     $QUADRATIC
     [[LOCAL ...;
       ...;
       ...]]
       $

 TFMREF>locale 1
     [[LOCAL A,B,C,ROOT1,ROOT2;
       LOOP:
         ...;
       ...;
       ...]]
       _
         TFMREF>locale 1
     LOCAL A,B,C,ROOT1,ROOT2;

 TFMREF>INFO
     5/25/79  23:29:21
     PROGRAM: QUAD.INT
     DOMAIN: SIMAL  PP                {transformations removed}
     INSTANCE: .PROGRAM QUADRATIC     {instance and locale different}
               $QUADRATIC
               ...
                 $
               .END

     LOCALE: LOCAL A,B,C,ROOT1,ROOT2;
 TFMREF>HARDCOPY                      {make file of prettyprinted instance}
 Output File>QUAD                     {no default extension}
     QUAD created

 TFMREF>EXIT                          {get out of TFMREF to Draco}
     Do you want to save the changes ? (Y/N)> Y  {SAVE check}
     QUAD.INT saved
 DRACO>
```

CHAPTER 8 USING THE PROGRAM TRANSFORMATION MECHANISM

The form and power of the transformations is discussed in the chapter on the transformation library generator (XFMGEN). The

TFMREF subsystem assumes that the transformation library already exists. The transformation library is loaded only when it must be to save working room in memory. Everytime the TFMREF system performs a transformation it prints out the before and after versions of the program fragment and will request a user OK if desired. The subsections below present the transformation related commands.

8.1. The SUGGEST command

SUGGEST

The SUGGEST command causes the system to examine its internal form of the current locale and suggest what transformations it would apply or look at. The suggest command goes hand-in-hand with the automatic suggestion of transformations option of the PARSE subsystem. If the automatic suggestion option was not taken then initially the suggest command will not be able to suggest any transformations until one is performed manually.

Even if you intend to perform transformations one at a time manually it is a good strategy to use the automatic suggestion of transformations option and then use the TRANSFORM command discussed below to strip out all the transformations which don't really apply (see the discussion of how transformations are suggested by PARSE).

8.2. The APPLY Command

```
                    APPLY <transformation name>
```

The APPLY command applies a specified transformation to the current locale. Remember by locale here we mean only the root of the internal form tree described from locale commands. Using APPLY is very tedious and it is not recommended. It is included simply because most transformation systems in the past have used such a command extensively. The TRANSFORM command discussed below is the recommended replacement for the APPLY command.

8.3. The TRANSFORM Command

TRANSFORM <code>-<code> TRANSFORM [LO CODE]<code> [HI CODE]<code> [APPROVAL MODE]<answer>

The TRANSFORM command automatically applies transformations of a certain application code range all over the selected locale. It allows the System Specialist to have the system request approval of each transformation (<answer> is ASK or NOASK). The transformations are applied to the locale from the leaves of the internal form up to the root of the locale. At each node the transformation with the highest application code is applied first.

When a transformation is successfully applied if the information in the metarules specifies transformations on the new program fragment then the bottom-up process begins again on the new fragment. It suffices to say that all the information in the metarules is taken advantage of by TRANSFORM. After a TRANSFORM the SUGGEST command will give rules suggested by the metarules during the TRANSFORM whose application codes were outside the range given to TRANSFORM. The more obscure transformations are usually suggested by the system in this way.

8.4. The ?TRANS Command

```
                    ?TRANS <transformation name>
```

The ?TRANS command prints out the text of a transformation. This is useful for examining a transformation suggested by the SUGGEST command. The format of the transformation is the same as for a catalog listing (see catalog).

8.5. The ?CLASS Command

```
                        ?CLASS <class name>
```

The ?CLASS command prints out the prefix keywords which match a class. This is only included because the ?TRANS command may print out a class name as part of the transformation and an initiate of the domain may want to remember what is included in the class.

8.6. The UNLOAD-TRANSFORM Command

```
                             UNLOAD-TRANSFORM
```

When the transformations are loaded notice is given to the user. As mentioned before the transformations are loaded automatically when required. However if the System Specialist decides to perform some refinements and he wants more room in memory he may remove the transformations from memory with the TFMUNLOAD command. They will be automatically loaded again if needed.

8.7. Transformation Example

The example given below shows a fragment of a session in which transformations are applied. Following the example session a prettyprinted version of the locale being refined is given. It is included so that the reader may compare the initial code and the final code. This example is an intermediate result in the refinement of the ATN generator in [Neighbors80].

```
TFMREF>PP
(DE GWOODS (gendict gentree)  (PROG (genrslt genrstack)
   (SETQ gentree (LIST gentree)
   (GO 'STREE)
   (NILL       -----------)
   STREE
   (OR (AND T T) (GO alab))
   (PUSH genrstack 'rlab)
   (PUSH gentree)
   (GO SUBJECT)
   rlab
   (PUSH genrstack 'rlab1)
   (PUSH gentree)
   (GO VERB-PHRASE)
   rlab1
   (GO S1)
   alab
   (NILL       -----------)
   (GENERR)
   S1
   (OR (AND (EQUAL 'QUESTION))
       (GO alab1))
   (POP gentree)
   (GO exit)
   alab1
   (NILL       -----------)
   (OR (AND (EQUAL 'DECLARE))
       (GO alab3))
   (POP gentree)
   (GO exit)
   alab3
   (NILL       -----------)
   (GENERR)
   exit
   (OR genrstack (RETRUN genrslt))
   (GO (POP genrstack))))

TFMREF>TRANSFORM (LO CODE) 50 (HI CODE) 90 (APPROVAL MODE) ASK
{ The System Specialist wishes to apply all transformations with
  application codes in the range 50 to 90. Approval for each
  transformation is also desired. The transformation library is loaded.
  If there were no .TLB file available an error message would be
  displayed and the process aborted. }

NOTE: transformations loaded

{ In the following lines Draco asks the System Specialist if it should
  apply each transformation. For each transformation Draco gives the
  following: <name>:<application code> <lhs> => <rhs>. If the specialist
  confirms the transformation (Y), the left-hand-side (lhs) is substituted
  by the right-hand-side (rhs) at that point in the locale. }

    NILL1: 50  (NILL       -----------)  =>  NIL  (Y/N)> Y
    ANDSEQT: 80  (AND T)  =>    (Y/N)> Y
    ANDSEQT: 80  (AND T)  =>    (Y/N)> Y
    ANDEMPTY: 80  (AND  )  =>  T  (Y/N)> Y
    ORT: 80  (OR T (GO alab))  =>  T  (Y/N)> Y

{ The following transformation relates large lhs and rhs expressions.
  Look for the "=>" delimiter. Remember Draco is printing the instance in
```

```
   the problem at hand - no the transformation definition. }

     PROGSEQT: 85  T(PUSH genrstack '...)
              (PUSH gentree )
              (GO SUBJECT)
              rlab
     (PUSH genrstack '...)
     (PUSH gentree)
     (GO VERB-PHRASE)
     rlab1
     (GO S1)
     alab
       =>  (PUSH genrstack '...)
           (PUSH gentree)
           (GO SUBJECT)
           rlab
           (PUSH genrstack '...)
           (PUSH gentree)
           (GO VERB-PHRASE)
           rlab1
           (GO S1)
           alab
               (Y/N)>Y
  { By successively applying NILL1 and PROGSEGNIL, the NILL
    comments will be eliminated from the locale. }
     NILL1: 50  (NILL       -----------)  =>  NIL  (Y/N)> Y
     PROGSEQNIL: 85  NIL  =>    (Y/N)> Y
     AND1: 80  (AND (EQUAL  'QUESTION))  =>  (EQUAL
                                 'QUESTION)  (Y/N)> Y
     NILL1: 50  (NILL       -----------)  =>  NIL  (Y/N)> Y
     AND1: 80  (AND (EQUAL  'DECLARE)  =>  (EQUAL
                               'DECLARE)  (Y/N)> Y
     NILL1: 50  (NILL       -----------)  =>  NIL  (Y/N)> Y
     PROGSEQNIL: 85  NIL  =>    (Y/N)> Y
     PROGSEQNIL: 85  NIL(OR (EQUAL  'DECLARE)
                    (GO alab3))
                      (POP gentree)
                      (GO exit)
                      alab3
                      =>    (OR (EQUAL  'DECLARE) (GO alab3))
                                (POP gentree)
                                (GO exit)
                                alab3
                      (Y/N)> Y
     PROGSEQNIL: 85  NILSTREE
     ...
     ...
     ...
     alab
     (GENERR)
     S1
     (OR ... ...)
     ...
     ...
     ...
     alab1
     (OR ... ...)
     ...
     ...
     ...
     alab3
     (GENERR)  =>  STREE
           ...
           ...
           ...
           alab
           (GENERR)
           S1
           (OR ... ...)
           ...
           ...
           ...
           alab1
           (OR ... ...)
           ...
           ...
```

```
          ...
          alab3
          (GENERR)    (Y/N)>Y

{ At this point the system specialists interrogates the
  system about the functions available. }

TFMREF>?
One of the following:
?CLASS  ?TRANS  ANNOTATE       APPLY   DOMAIN  EXIT    HARDCOPY
HELP    INFO    INSTANCE       LOCALE  NOINSTANCE      PP
REFINE  REFLRU  SAVE     SUGGEST TACTICS TRANSFORM
UNLOAD-REFINE UNLOAD-TRANSFORM

{ The pretty-printed version of the locale given below
  shows the new version of the code after the previous
  transformations were applied. }

TFMREF>PP
(DE GWOODS (gendict gentree)  (PROG (genrslt genrstack)
   (SETQ gentree (LIST gentree)
   (GO 'STREE)
   STREE
   (PUSH genrstack 'rlab)
   (PUSH gentree)
   (GO SUBJECT)
   rlab
   (PUSH genrstack 'rlab1)
   (PUSH gentree)
   (GO VERB-PHRASE)
   rlab1
   (GO S1)
   alab
   (GENERR)
   S1
   (OR (EQUAL 'QUESTION)
       (GO alab1))
   (POP gentree)
   (GO exit)
   alab1
   (OR (EQUAL 'DECLARE)
       (GO alab3))
   (POP gentree)
   (GO exit)
   alab3
   (GENERR)
   exit
   (OR genrstack (RETRUN genrslt))
   (GO (POP genrstack))))
```

CHAPTER 9
USING THE TFMREF REFINEMENT SUBSYSTEM

9.1. The TFMREF Commands Which Work With Refinements

9.2. How a Component is Used

This section discusses how the fields of a component are used in the refinement process to choose an implementation for the operation or object the component represents.

First, the IOSPEC conditions on the component should be verified by examining the internal form or refinement history of the surrounding internal form of the node to be refined. Restrictions on the legal internal forms accepted by the domain language parser might make this step easier.

Second, a REFINEMENT is chosen and the refinement CONDITIONS are checked. If an implementation decision condition is violated then the refinement may not be used. Local conditions on the domain objects are formed into surrounding code for the refinement body. The hope is that transformations for the domain will be able to remove this surrounding code by "proving" the conditions correct and removing the code.

The user is then asked about any ADJUSTMENTS for the refinement. If the user supplies no adjustments then the default adjustments are used.

Next, the refinement body is instantiated into the internal form according to the users wishes for INSTANTIATION and the allowed instantiations for the refinement. The body is instantiated with minimal renaming to avoid naming conflicts. If the refinement is instantiated as a function and a function already exists then the already defined function is used.

Once the refinement is inserted, any necessary RESOURCES are added to the initialization phase of the developing program. These resources are usually high-level program fragments which also have to be refined.

Finally, the ASSERTIONS for the refinement are made in the scope of the domain instance. The assertions are a kind of lock and key mechanism with the conditions of other refinements. When two domain instances are merged into a single instance of a same or other domain, then the assertions are checked for consistency. This places the overly strong restriction that all objects in a domain of the same type have the same implementation. More experience with domains could probably remove this restriction. If the asserted conditions conflict, then the refinement of the program must be backed up.

9.3. The Refinement Mechanism

The refinement mechanism of Draco applies the component library of a domain to a locale within an instance of the domain in the internal form tree for the program being refined. The locale is bounded by a domain instance which is a part of the internal form tree in the internal form of a particular domain. Refinements are made in one domain at a time on an instance of the domain. The locale mechanism is important for refinements in that the "inner loop" of the program should be refined first to pick efficient implementations. These implementation decisions will affect the choices outside of the inner loop through the assertion and condition mechanism of the components.

The Draco refinement mechanism applies the components to the locale internal form tree using application policies similar to transformation application policies. In general, top-down application is the best policy to avoid conflicting conditions which would require a backup of the refinement.

From the previous discussion about the selection of a refinement for a component and the user interaction necessary to make a choice, it is evident that the user needs some mechanism to keep Draco from asking too many questions. The user needs the ability to specify guidelines for answering the questions and these guidelines are called "tactics."

The TACTICS subsystem (see Chapter 10) of Draco allows the user to interactively define tactics which answer refinement questions for the refinement mechanism. The subsystem also allows the user to read and write tactics from storage. A standard set of tactics is already available. When the refinement mechanism requires a user response, it first applies the tactics to see if one of them provides an answer.

The refinement user interface could be used for applying refnements one at a time. This would be very tedious work, as tedious as applying transformations one at a time. In general, early versions of a high-level, domain-specific program are refined by the default tacts. These use easy and uncomplicated default refinements to obtain a first implementation and to check whether the system implements everything the user desires.

9.4. The TFMREF Command : REFINE

Under the Transformation and Refinement subsystem TFMREF, the REFINE command can be used to invoke the Refinement User Interface. This is a new set of commands that enable the user to perform refinements and provide tactics. Alternatively the Refinement User Interface is activated if the user interrupts automatic tactics during refinement.

9.5. Commands available through the Refinement User Interface

The following sections describe the low-level Refinement commands that are accessible through the Refinement User Interface.

9.5.1. The TRY command

```
TRY [REFINEMENT NUMBER] <ref> [UNDER INSTANTIATION] <inst>
```

The TRY command attempts to apply the selected refinement of a component by it's order number (<ref>) in the component. The default refinement is always number 0. The instantiation (<inst>) is either INLINE or FUNCTION. INLINE instatiates the refinement inline as a macro-expansion. FUNCTION abstracts a function of the refinement and inserts a call to the abstracted function. With TRY the user is asked for final approval.

9.5.2. The USE command

```
   USE [REFINEMENT NUMBER] <ref> [UNDER INSTANTIATION] <inst>
```

The USE command is equivalent to the TRY command except that the user's approval is not requested.

9.5.3. The DEFER command

```
 DEFER
```

The effect of the DEFER command us similar to that of ABORT, that is it interrupts the refinement process; but in this case it defers making a decision about the component refinement and shifts control to the *entry* tactics.

9.5.4. The ABORT command

```
 ABORT
```

The ABORT command aborts the refinement process and transfers control back to the TFMREF subsystem level. Tactics are halted.

9.5.5. The DO command

```
DO [TACTICS-COMMAND] <tactic *CMD* name>
```

The DO command produces a search of the tactics list for the given tactic. If the list is not empty, it executes the associated command group.

9.5.6. The HELP command

```
      HELP
```

The HELP command prints a summary of the commands available through the Refinement User Interface. The printed text is read from the file REFUSER.HLP. Thus, it can be customized, if necessary, by modifying the file. The current file contents is given in section 9.5.8.

9.5.7. The INFORMATION command

The INFORMATION command enables the System Specialist to accquire information on Assertions and the use of memory by the system. Thus, there are two different formats:

```
      INFORMATION [ABOUT] ASSERTIONS [IN] <domain specification>
```

and

```
      INFORMATION [ABOUT] MEMORY-USAGE
```

The first form accepts the following for <domain spoecification>:

```
      ALL-DOMAINS
      CURRENT-DOMAIN
      DOMAIN (NAMED) <domain name> (ON) <objects and operations>
```

in each case the user gets a list of the relevant assertions. The assertions provide the implementing concept information (e.g., SIMAL integers as LISP numbers). The second for reports the amout of free memory remaining as in:

```
      Free Storage: NNNN  Full-word Space: MMMM
```

9.5.8. A summary of the Refinement User Interface commands

The HELP (see sections 9.5.6) command provides the following information.

```
REFINE>HELP
The Refinement User Interface Commands are:

  TRY <refinement> <instantiation>
        attempt to use a refinement - show and ask before use
  USE <refinement> <instantiation>
        use a refinement and don't show or ask
  DEFER
        defer control back to the *entry* tactics - restart tactics
  ABORT
        return to the TFMREF subsystem level - stop tactics
  DO <tactic *CMD* name>
        do a predefined tactic command
  HELP
        this listing
All of these commands are discussed in more detail in the
Draco user manual.
```

9.6 An example session with REFINE

This section presents a session with REFINE. In the session don't be concerned with the meaning of the domain language being manipulated (it is the same one used in the example of Chapter 8). What is of concern in the example is the way in which the commands of the Refinement User Interface interact with the user to manipulate this example. In the transcript, user responses are underlined and comments included between curly brackets. In particular, every time *{ .... }* appears, it means that the output from Draco is similar for different components, and it is not transcribed in order to prevent the example from being excessively long. The transcript follows:

```
TFMREF>?
One of the following:
?CLASS  ?TRANS  ANNOTATE       APPLY   DOMAIN  EXIT    HARDCOPY
HELP    INFO    INSTANCE       LOCALE  NOINSTANCE      PP
REFINE  REFLRU  SAVE     SUGGEST TACTICS TRANSFORM
UNLOAD-REFINE UNLOAD-TRANSFORM
TFMREF>tactics
*NOTE: file DEMO.TCT accessed from Draco disk area.
**********************
Parse Completed 0 errors detected
**
Parse Completed 0 errors detected
TFMREF>domain gen                         { work in GEN domain }
NOTE: DRACO domain being removed
TFMREF>instance                           { outline of a GEN instance }
GENERATOR GWOODS
NETWORK STREE
;------------------------------------------------------------
STREE...
S1...
.END
    (Y/N)>Y                             { accept this instance }
TFMREF>pp                                 { prettyprint of GEN instance }
GENERATOR GWOODS
NETWORK STREE
;------------------------------------------------------------
STREE  S1  | none              | gen SUBJ at SUBJECT
   |                   | gen VP at VERB-PHRASE
;------------------------------------------------------------
S1     exit| TYPE='QUESTION    | out "?"
;      -----------------------------------------------------
       exit| TYPE='DECLARE     | out "."
;------------------------------------------------------------
.END
TFMREF>noinstance
NOTE: no domain instance selected
TFMREF>refine
NOTE: component library index loaded
NOTE: new domain instance automatically selected
```

```
COMPONENT: DNAME
PURPOSE:
Represent the given name as a data item rather than as
a variable representing a value.
STREE                                    { how this component appears in program }
REFINEMENT: quote the name for LISP
DOMAIN: LISP

COMPONENT: COMMENT
PURPOSE: To represent comments from the ATN domain
;------------------------------------------------------------
REFINEMENT: LISP comment mechanism
BACKGROUND:
This is an in-memory comment, perhaps a comment scanned
off by a LISP read would be better.
DOMAIN: LISP

COMPONENT: NOTEST
PURPOSE: No arctest is performed; thus, the test always succeeds.
                                 { concept does not print in domain }
REFINEMENT: use a LISP true for the test
DOMAIN: LISP

COMPONENT: NOTEST
PURPOSE: No arctest is performed; thus, the test always succeeds.

REFINEMENT: use a LISP true for the test
DOMAIN: LISP

COMPONENT: TEST-SEQ
PURPOSE:
Try test1; if it succeeds then try test2; otherwise fail the test.

REFINEMENT: use McCarthy LISP AND for test sequence
DOMAIN: LISP
                                 { user interrupts tactics from keyboard }
NOTE: tactics interrupted by user
NOTE: tactics have failed to find a refinement
                                 { invokes interactive refine interface }
User Refinement Interface
REFINE>do summary                        { interpret *CMD*.SUMMARY tactic }
Component Summary
COMPONENT: TEST-SEQ
PURPOSE:
Try test1; if it succeeds then try test2; otherwise fail the test.
The component appears in the program as:
The refinements for the component are:
REFINEMENT: use McCarthy LISP AND for test sequence
INSTANTIATION: INLINE
DOMAIN: LISP

REFINE>help

The Refinement User Interface Commands are:

  TRY <refinement> <instantiation>
attempt to use a refinement - show and ask before use
  USE <refinement> <instantiation>
use a refinement and don't show or ask
  DEFER
defer control back to the *entry* tactics - restart tactics
  ABORT
return to the TFMREF subsystem level - stop tactics
  DO <tactic *CMD* name>
do a predefined tactic command
  HELP
this listing
All of these commands are discussed in more detail in the
Draco user manual.

REFINE>information (ABOUT) memory-usage
Free Stoarge: 40836  Full-Word Space: 6488

REFINE>defer                             { restart tactics }
```

```
COMPONENT: TEST
PURPOSE: Try tests for arc
REFINEMENT: use McCarthy LISP AND for test sequence
DOMAIN: LISP

COMPONENT: DNAME
PURPOSE:
Represent the given name as a data item rather than as
a variable representing a value.
SUBJ                                    { how this component appears in program }
REFINEMENT: quote the name for LISP
DOMAIN: LISP

COMPONENT: GTFETCH
PURPOSE:
Get the subtree associated with the selector at the top
level of the tree being generated
REFINEMENT: check tree existence with LISP OR
DOMAIN: LISP
NOTE:function GEN.GTFETCH.0 defined      { refinement instantiated a function }

{..........}

                                { user interrupts tactics from keyboard }
NOTE: tactics interrupted by user
NOTE: tactics have failed to find a refinement
                                { invokes interactive refine interface }
User Refinement Interface
REFINE>information (ABOUT) assertions (IN) all-domains
GEN actions as LISP_inline_sequence
            (GEN/ACTION-SEQ/execution sequence in a LISP PROG)
GEN arcs as LISP_inline_sequence
            (GEN/GARC/use a LISP COND)
GEN states as LISP_inline_sequence
            (GEN/GCALL/simulate the call in LISP)

REFINE>do summary
Component Summary
COMPONENT: GSTATE
PURPOSE: define a state in the generator network

The component appears in the program as:
STREE
The refinements for the component are:
REFINEMENT: as inline LISP
ASSERTIONS: GEN states as LISP_inline_sequence
INSTANTIATION: INLINE
ASSERTIONS: GEN states as LISP_inline_sequence   { *CMD*.SUMMARY error }
DOMAIN: LISP

REFINE>defer                            { restart tactics }

COMPONENT: DNAME
PURPOSE:
Represent the given name as a data item rather than as
a variable representing a value.
TYPE                                    { how this component appears in program }
REFINEMENT: quote the name for LISP
DOMAIN: LISP

COMPONENT: GVFETCH
PURPOSE:
Get the value of the subtree associated with the selector
at the top level of the tree being generated
REFINEMENT: extract value with LISP COND
DOMAIN: LISP
NOTE: function GEN.GVFETCH.0 defined

COMPONENT: QUOTE
PURPOSE: Put a literal name in an ATN tree
'QUESTION
REFINEMENT: use a LISP quoted atom
ASSERTIONS: ATN trees as LISP_lists
DOMAIN: LISP

{..........}
```

```
COMPONENT: GSTATE
PURPOSE: define a state in the generator network
S1
REFINEMENT: as inline LISP
ASSERTIONS: GEN states as LISP_inline_sequence
DOMAIN: LISP

NOTE: tactics interrupted by user
NOTE: tactics have failed to find a refinement

User Refinement Interface
REFINE>abort                              { exit tactics }
NOTE: aborting the refinement process

TFMREF>instance                           { still working in GEN domain }
GENERATOR GWOODS
NETWORK
STREE...
.END
    (Y/N)>Y

TFMREF>pp                                 { inside-out tactics have created LISP }
GENERATOR GWOODS
NETWORK <<LISP>>
STREE   <<LISP>>
.END

TFMREF>noinstance
NOTE: no domain instance selected

TFMREF>refine
NOTE: new domain instance selected automatically

COMPONENT: GSTATE
PURPOSE: define a state in the generator network
STREE
REFINEMENT: as inline LISP
ASSERTIONS: GEN states as LISP_inline_sequence
DOMAIN: LISP

COMPONENT: STATES-SEQ
PURPOSE:
Specify the ordering of the states in the original description.
REFINEMENT: keep the same ordering in LISP inline
ASSERTIONS: GEN states as LISP_inline_sequence
DOMAIN: LISP

{..........}

NOTE: refinement replaced a domain       { last bit of GEN instance refined }
NOTE: new domain instance selected automatically
                                 { another GEN instance was found }

{..........}

NOTE: no domain instance selected        { no more GEN domain was found }

TFMREF>domain lisp
NOTE: GEN domain being removed
TFMREF>instance
(SETQ genrslt (NCONC genrslt (LIST item)))
            (Y/N)>N              { not LISP domain instance we want }
(COND [(... ...)]
      [(... ...)])  (Y/N)>N              { still not the instance }
(OR (MEMB selector (CAR gentree))
    (ASSOC selector (CAR gentree))
    (GENERR))       (Y/N)>N              { yet again not the instance }
(DE GWOODS (gendict gentree) (PROG (genrslt genstack)
     (... ... ...)
     (... ...)
     ...
     ...
     ...
     exit
```

```
      (OR ... ...)
      (... ...)))    (Y/N)>Y              { yes select this LISP instance }

 TFMREF>pp
 (DE GWOODS (gendict gentree)  (PROG (genrslt genrstack)
    (SETQ gentree (LIST gentree)
    (GO 'STREE)
    (NILL       -----------)
    STREE
    (OR (AND T T) (GO alab))
    (PUSH genrstack 'rlab)
    (PUSH gentree)
    (GO SUBJECT)
    rlab
    (PUSH genrstack 'rlab1)
    (PUSH gentree)
    (GO VERB-PHRASE)
    rlab1
    (GO S1)
    alab
    (NILL       -----------)
    (GENERR)
    S1
    (OR (AND (EQUAL 'QUESTION))
        (GO alab1))
    (POP gentree)
    (GO exit)
    alab1
    (NILL       -----------)
    (OR (AND (EQUAL 'DECLARE))
        (GO alab3))
    (POP gentree)
    (GO exit)
    alab3
    (NILL       -----------)
    (GENERR)
    exit
    (OR genrstack (RETRUN genrslt))
    (GO (POP genrstack))))

 TFMREF>save
 NOTE: WG.INT saved

 { At this point the transformations shown in the example of
   Chapter 8 should be applied to remove unused generality.
   However this is not required. }

 { All that is required now is to instantiate Draco's model
   of functions and system phases for LISP. This is held in
   the Draco domain so that the refinement mechanism knows
   how to create functions and phases }

 TFMREF>domain draco
 NOTE: LISP domain being removed
 TFMREF>refine
 NOTE: file DRACO.RLB accessed from Draco disk area
 NOTE: component library index loaded
 NOTE: new domain instance selected automatically

 { See the example in Chapter 5, where
   another subset of these Draco components
   is used to build the Component Library DRACO.RLB }

 COMPONENT: *APARAMS-SEQ*
 PURPOSE: The sequence of actual parameters of a Draco function call
 (...,)
 REFINEMENT: LISP actual parameters
 BACKGROUND: The actual parameters are treated as an
     execution sequence.
 DOMAIN: LISP

 COMPONENT: *APARAMS*
 PURPOSE: An actual parameter of a Draco function call
 ()
 REFINEMENT: LISP actual parameters
```

```
BACKGROUND: The actual parameters are evaluated
DOMAIN: LISP

{..........}

DOMAIN: DRACO
NOTE: refinement merged a domain
note: no instance of DRACO domain in locale of refinement
{ Draco looks for other Draco domain instances and finds one }
{ the top level with all the created functions and temps }
The modules are:
       GEN.GOUT.O(item){}[]
       GEN.GVFETCH.0(selector){TEMP}[]
       GEN.GTFETCH.0(selector){gentree}[]
       DRACO.START.0(){alab,alab0,alab2,exit,rlab,rlab0}
              [alab,alab1,alab3,exit,rlab,rlab1]
    (Y/N)>Y
TFMREF>refine

COMPONENT: *FPARAMS-SEQ*
PURPOSE: The formal parameter sequence of a Draco function definition
(...selector)
REFINEMENT: LISP formal parameters
DOMAIN: LISP

COMPONENT: *FPARAMS*
PURPOSE: A formal parameter of a Draco function definition
()
REFINEMENT: LISP formal parameters
DOMAIN: LISP

{..........}

NOTE: refinement replaced Draco top level
TFMREF>domain lisp
NOTE: DRACO domain being removed

TFMREF>pp
(DE GOUT (item) (PROG ()
    (RETURN (SETQ genrslt (NCONC
                           genrslt
                           (LIST
                            item))))))
(DE GVFETCH (selector) (PROG (TEMP)
     (RETURN (COND [(ATOM (SETQ TEMP (OR (MEMB
                                  selector
                                  (CAR
                                   gentree))
                                 (ASSOC
                                  selector
                                  (CAR
                                   gentree))
                                 (GENERR))))]
           [(CADR TEMP)]))))
(DE GTFETCH (selector) (PROG (gentree)
     (RETURN (OR (MEMB selector (CAR gentree))
         (ASSOC selector (CAR gentree))
         (GENERR)))))
(DE START () (PROG (alab alab0 alab2 exit rlab rlab0)
     (RETURN (DE GWOODS (gendict gentree)
       (PROG (genrslt genrstack)
           (SETQ gentree (LIST gentree))
           (GO 'STREE)
           STREE
           (PUSH genrstack 'rlab)
           (PUSH gentree (GTFETCH
                          'SUBJ))
           (GO SUBJECT)
           rlab
           (PUSH genrstack 'rlab)
           (PUSH gentree (GTFETCH
                          'VP))
           (GO VERB-PHRASE)
           rlab1
           (GO S1)
           alab
```

```
            (GENERR)
            S1
            (OR (EQUAL (GVFETCH 'TYPE)
                 'QUESTION)
                (GO alab1))
            (GOUT "?")
            (POP gentree)
            (GO exit)
            alab1
            (OR (EQUAL (GVFETCH 'TYPE)
                 'DECLARE)
                (GO alab3))
            (GOUT ".")
            (POP gentree)
            (GO exit)
            alab3
            (GENERR)
            exit
            (OR genrstack
                (RETURN genrslt))
            (GO (POP genrstack)))))))

TFMREF>hardcopy wg
NOTE: WG.DOC created

TFMREF>exit
Do you want to save the changes ? (Y/N)> Y
NOTE: WG.INT saved
DRACO>
```

This example is somewhat unrealistic in that it takes a fragement of the GEN domain and refines it all the way to an executable in LISP. GEN is a modelling domain for NLP/RDB which glues together a collection of domains (e.g., DIC, GEN, ATN, RDB) to perform natural language database query (see [Neighbors80]). Because of this the START function simply returns the definition of the GEN fragment and the exception (GENERR) and output (GOUT) handlers are not connected. However this small example demonstrates the flavor of using the Draco refinement mechanism.

CHAPTER 10
USING THE TFMREF TACTICS SUBSYSTEM

The TACTICS subsystem's objective is to provide "guidelines" that can be used during the refinement process to prevent Draco for asking too many questions. The TACTICS subsystem in an interpreter that allows the user to define tactics interactively (DEFINE) or to use an existing tactic set (LOAD).

The TACTICS subsystem is called from the TFMREF system with the command TACTICS. This activates the TACTICS interpreter. After the loading of the interpreter (* are printed during this process) we can use any of the TACTICS commands.

```
TFMREF>TACTICS

**HELP;                                 { prints the help file }
The TACTICS commands are:
 DEFINE - define a tactic
 LIST   - list the tactics to screen or file
 DELETE - delete a tactic
 LOAD   - load tactics from a file
 HELP   - this list
 EXIT   - return to TFMREF subsystem

More detail on the syntax of these commands may be
found in the Draco manual. Remember, all commands must
be terminated by a semicolon.

**LOAD DEMO;                            { DEMO.TCT is the tactic file }
**************************              { each * is one line processed }
Parse Completed 0 errors detected
*EXIT;

TFMREF>
```

The DEFINE command defines the rules. The format of the command is

```
     DEFINE <rulegroup-name>.<rule-name> = <rule>
```

Rules with rule-name "*ENTRY*" are automatically run as tactics. Rules with the rulegroup-name "*CMD*" must be invoked from the Refinement User Interface. The goal of all tactic rules is to get a refinement for a single component. In the DEFINE command we can determine which fields of the component we would like to display. The simple demonstration tactics DEMO.TCT we used for out refinement session in the previous chapter are shown below.

```
DEFINE HEAD.*ENTRY* = "
";

DEFINE DEMO.*ENTRY* = COMPONENT,PURPOSE
              LOC4,
              [ALL,REFINEMENT
                   CONDITIONS,ASSERTIONS,
                   BACKGROUND,DOMAIN],
              [ALL<DIRECTIVE>,USE]
              [ALL<FUNCTION INSTANTIATION>,USE FUNCTION],
              [ALL<INLINE INSTANTIATION>,USE INLINE];

DEFINE *CMD*.SUMMARY = "Component Summary",
               COMPONENT,PURPOSE,
               IOSPEC,DECISION,
                "The component appears in the program as:",
               LOC 10,
                "The refinements for the component are:",
               [ALL,REFINEMENT,CONDITIONS,ASSERTIONS,
                BACKGROUND,DIRECTIVE,
                INSTANTIATION,ASSERTIONS,
                RESOURCES,ADJUSTMENTS,
                DOMAIN];
EXIT
```

When the refinement mechanism needs to choose a refinement for a component it runs the *ENTRY* tactics in the order in which they were defined. The HEAD tactics in this case simply prints a CRLF but does not choose a refinement. The refinement mechanism then moves to the DEMO tactics that first print the component context to a internal tree depth of 4 (LOC 4). The [ALL...] construct steps though each refinement for the component until a refinement has been selected. The first [ALL...] construct simply displays the fields of the component. No refinement is found so processing advances to the next [ALL...] construct. The [ALL<DIRECTIVE>...] is met for refinements that contain a directive from the Domain Designer. These are seldom used suggestions that usually result in a refinement DEFER command. These occur in special cases where the inside-out refinement implies that this component will be removed by refinement and as such no refinement is necessary. The [ALL<FUNCTION..] tactic states that if we have already refined a function for this component then just use a call to that function. The [ALL<INLINE..] tactic states that if we can inline the refinement then use it inline. Thus, the DEMO.*ENTRY* tactics will create a function where possible - resulting in smaller but slower code.

The results of the *CMD*.SUMMARY tactics can be seen in the previous Chapter where the REFINE>DO SUMMARY command was given. They print inforformation (print ASSERTIONS twice - an error) but make no USE or TRY decisions. They are support for the System Specialist making decisions. If none of the tactics with *ENTRY* find a refinement then the Refinement User Interface is invoked and the user may use the *CMD* rules for information and making refinement choices.

Contrary to the DEMO.*ENTRY* rule above, the INLINE.*ENTRY* rule below will inline as much as possible and as a last resort use the default refinement. Heavy inlining makes rich opportunities for transformations and results in faster programs.

```
DEFINE HEAD.*ENTRY* = COMPONENT,LOC 3;

DEFINE INLINE.*ENTRY*=[ALL<DIRECTIVE>,USE],
              [ALL<AVAILABLE FUNCTION>,USE FUNCTION],
              [ALL<FUNCTION INSTANTIATION>,USE FUNCTION],
              USE DEFAULT;

EXIT
```

The tactics parser and prettyprinter are in Appendices VII and IX.

NOTES

[NOTE 1]	Draco is Latin for dragon.
[NOTE 2]	SADT ® is a registered trademark of SofTech Inc.
[NOTE 3]	Members of group 4800 can access the software.
[NOTE 4]	We would have liked to have made the Draco menu driver compatible with the standard Tops-20 Exec but in UCI Lisp under the PA1050 simulator we have input activation only on linefeed, carriage return, and escape. These are the control keys used by the menu driver.
[NOTE 5]	Draco uses "=>" to denote implication for transformations.
[NOTE 6]	We plan to change from using numbers to domain-specific names for the operations.
[NOTE 7]	The EXAMINE subsystem and history recording are not operational in the current system.
[NOTE 8]	In fact the scheme is actually iterative to avoid very large levels of recursion resulting from printing large programs where the statements are in a tree. This is necessary to avoid blowing the LISP special and regular pushdown lists.

REFERENCES

[Aho79]	Aho, A.V., and Ullman, J.D., **Principles of Compiler Design**, Addison-Wesley Publishing Co., 1977
[Neighbors80]	Neighbors, J.M., **Software Construction Using Components**, PhD thesis, University of California at Irvine, 1980, UCI ICS technical report TR-160.
[Schorre64]	Schorre, D.V., **META II: A Syntax-Oriented Compiler Writing Language**, In Proceedings of the ACM National Conference, pages D1.3-1 to D1.3-11. ACM, 1964.
[Standish76]	Standish,T.A., Harriman, D.C., Kibler, D.F., and Neighbors, J.M., **The Irvine Program Transformation Catalogue**, Technical Report, ICS Dept., University of California at Irvine, January 1976.

APPENDIX I
A COMPLETE EXTERNAL/INTERNAL LANGUAGE DEFINITION

In this appendix the complete external and internal definition for an example language called SIMAL is given along with an example program written in SIMAL. SIMAL represents a conventional ALGOL-like language. It is hoped that domain languages will differ greatly from this form.

I.1 External/Internal SIMAL Definition

The following is the file SIMAL.DEF. If any errors occur during the parsing of a SIMAL program, the rule names in the error messages will refer to parse rules in this file.

```
.DEFINE SIMAL
[ SIMAL simple ALGOL-like language for examples ]
[  James Neighbors -- Last Modified March 11, 1982 ]

SIMAL = ".PROGRAM"
    .TREE(PGM PGMSEQ
          NAME .TREE(AP APSEQ "(" EXP $("," EXP) ")")
              .NODE(PROCCALL #2 #1)
           $<1:?>FNDEF)
".END" ;

FNDEF = "$" NAME
  .TREE(FP FPSEQ "(" NAME $("," NAME) ")")
  BLOCK
"$" .NODE(FNDEF #3 #2 #1) ;

STMT = BLOCK /
       "IF" BEX "THEN" STMT
    ("ELSE" STMT .NODE(IFELSE #3 #2 #1) /
     .EMPTY .NODE(IF #2 #1) ) /
       "WHILE" BEX "DO" STMT .NODE(WHILE #2 #1) /
       "REPEAT" STMT "UNTIL" BEX .NODE(REPEAT #2 #1) /
       "FOR" NAME ":=" AEX
     ("STEP" AEX "TO" AEX "DO" STMT
                         .NODE(FOR #5 #4 #3 #2 #1) /
      .EMPTY     "TO" AEX "DO" STMT
                         .NODE(FOR1 #4 #3 #2 #1)   ) /
       "RETURN" (EXP .NODE(RETVAL #1) / .EMPTY .NODE(RETURN)) /
       "GO" "TO" ID .NODE(GOTO *) /
       "(" STMT ")" .NODE(PAREN #1) /
       SIMALFN /
       NAME
    ( "[" .TREE(SL SLSEQ EXP $("," EXP)) "]"
              .NODE(ASELECT #2 #1)
           ":=" EXP .NODE(SASSIGN #2 #1) /
      ":=" EXP .NODE(ASSIGN #2 #1) /
      ":" STMT .NODE(LABEL #2 #1) /
      .TREE(AP APSEQ "(" EXP $("," EXP) ")")
              .NODE(PROCCALL #2 #1) ) ;

BLOCK = "[[" ("LOCAL" .TREE(LOC LOCSEQ NAME $("," NAME)) ";" /
      .EMPTY .NODE(NOLOC))
     .TREE(BLK BLKSEQ STMT $(";" STMT)) "]]"
.NODE(BLOCK #2 #1) ;

NAME = ID .LITERAL ;

EXP = STRING .NODE(STRING *) / BEX ;

BEX  = BEX1 $("!" BEX1 .NODE(OR #2 #1)) ;
BEX1 = BEX2 $("&" BEX2 .NODE(AND #2 #1)) ;
BEX2 = "%" BEX3 .NODE(NOT #1) / BEX3 ;
BEX3 = "TRUE" .NODE(TRUE) / "FALSE" .NODE(FALSE) /
       AEX $("<=" AEX .NODE(LESSEQ #2 #1) /
     ">=" AEX .NODE(GTREQ #2 #1) /
     "<"  AEX .NODE(LESS #2 #1) /
     ">"  AEX .NODE(GTR #2 #1) /
     "="  AEX .NODE(EQUAL #2 #1) /
     "#"  AEX .NODE(NOTEQ #2 #1) ) ;

AEX  = AEX1 $("+" AEX1 .NODE(ADD #2 #1) /
      "-" AEX1 .NODE(SUB #2 #1) ) ;
AEX1 = AEX2 $("*" AEX2 .NODE(MPY #2 #1) /
      "//" AEX2 .NODE(IDIV #2 #1) /
      "/" AEX2 .NODE(DIV #2 #1) ) ;
AEX2 = AEX3 $("^" AEX2 .NODE(EXP #2 #1) ) ;
AEX3 = "+" AEX4 / "-" AEX4 .NODE(MINUS #1) / AEX4 ;
AEX4 = NUMBER .NODE(NUMBER *) /
       SIMALFN /
       NAME ( "(" .TREE(AP APSEQ EXP $("," EXP)) ")"
                 .NODE(FNCALL #2 #1) /
      "[" .TREE(SL SLSEQ EXP $("," EXP)) "]"
                 .NODE(SSELECT #2 #1) /
      .EMPTY ) /
       "(" BEX ")" .NODE(PAREN #1) /
       BLOCK ;

SIMALFN = "SQRT" "(" EXP ")" .NODE(SQRT #1) /
  "INT" "(" EXP ")" .NODE(INT #1) /
  "ABS" "(" EXP ")" .NODE(ABS #1) /
  "PRINT" "(" .TREE(PRINT PRSEQ EXP $("," EXP)) ")" /
  "READNUM" .NODE(READNUM) /
  "READCHAR" .NODE(READCHAR) /
  "READSTRING" .NODE(READSTRING) /
  "WRITENUM" "(" EXP ")" .NODE(WRITENUM #1) /
  "WRITECHAR" "(" EXP ")" .NODE(WRITECHAR #1) /
  "WRITESTRING" "(" EXP ")" .NODE(WRITESTRING #1) ;

PREFIX : SPACING ;
ID : SPACING .TOKEN ALPHA $<?:10>(ALPHA / DIGIT) .DELTOK ;
STRING : SPACING .TOKEN .ANY('") $.ANYBUT('") .ANY('") .DELTOK ;
NUMBER : SPACING .TOKEN $<1:?>DIGIT
   (.ANY('.) ($<1:?>DIGIT (EXPNT / .EMPTY) /
               .EMPTY) /
    EXPNT /
    .EMPTY ) .DELTOK ;
```

```
EXPNT : .ANY('E) (.ANY('+!'-) / .EMPTY) $<1:?>DIGIT ;
ALPHA : .ANY('A:'Z ! 'a:'z) ;
DIGIT : .ANY('0:'9) ;
SPACING : $.ANY(32!10!13!9) ;

.END
```

I.2. Example SIMAL Program

The familiar quadratic equation root solution.

```
.PROGRAM QUADRATIC

$QUADRATIC
 [[ LOCAL A,B,C,ROOT1,ROOT2;
    LOOP:
    PRINT("QUADRATIC EQUATION SOLVER");
    PRINT("INPUT A,B,C PARAMETERS ");
    A:=READNUM;
    IF A=0 THEN RETURN;
    B:=READNUM;
    C:=READNUM;
    ROOT1:=(-B+SQRT(B^2-4*A*C))/(2*A);
    ROOT2:=(-B-SQRT(B^2-4*A*C))/(2*A);
    PRINT("THE ROOTS ARE: ",ROOT1," AND ",ROOT2);
    GOTO LOOP ]]
$
.END
```

APPENDIX II
THE DEFINITION OF DRACO BNF IN DRACO BNF

This appendix presents the file PARGEN.DEF which describes Draco BNF in Draco BNF and the internal LISP form parsers take when they are constructed. When PARGEN cites a rule in an error during the construction of a domain parser the rule is from this file. Also included in this appendix is a catalog listing of the transformations (PARSER.TLB) which are used to optimize parsers (see parser optimization).

II.1. The File PARGEN.DEF

```
.DEFINE PARSER

[ PARGEN Main Parser Generator
  James Neighbors -- Last Modified May 7, 1983 ]

PARSER = ".DEFINE" PARSER1 ".END" .RESOLVE(RULE) ;

PARSER1 = ID .USE(RULE)
          .LIST(PROGN
                .NODE(DEFINE-PARSER-FN PARSE (LAMBDA NIL (PARSER-INITI
                $(ST / COMMENT)) ;

COMMENT = "[" CMNTCHRS "]" .NODE(NILL *) ;

ST = [[ ID .DEF(RULE) .LITERAL .MSG(.CR "Rule " * .COL(30))
        ("=" .LITERAL EX1 .NODE(PARSER-RULE #2 #1) /
         ":" TEX1 .NODE(PARSER-TOKEN #1))
        ";" .NODE(DEFINE-PARSER-FN #2 (LAMBDA NIL #1)) ]
      ERRST .NODE(NILL) ] ;
ERRST : .TOKEN $.ANYBUT(';) .ANY(';) .DELTOK ;

EX1 = .LIST(OR EX15 $("/" EX15)) ;

EX15 = EX2 ("|" EX17 .NODE(PARSER-BACKTRACK #2 #1) / .EMPTY) ;
EX17 = EX2 ("|" EX17 .NODE(PARSER-BACKTRACK #2 #1) / .EMPTY) ;

EX2 = EX3 .LIST(AND $EX3) .NODE(AND #2 (OR #1 (PARSER-ERROR))) ;

EX3 = ID .USE(RULE) .NODE(*) /
      STRING .NODE(PARSER-TEST-STRING *) /
      "(" EX1 ")" /
```

```
       "$" RPTPR /
       ".NODE" "(" NLIST ")" .NODE(PARSER-NODE #1) /
       ".LITERAL" .NODE(PARSER-LITERAL) /
       ".LITCHAR" .NODE(PARSER-LITCHAR) /
       ".EMPTY" .NODE(AND T) /
       ".TREE" "(" ID .USE(COMPONENT) .LITERAL
                   ID .USE(COMPONENT) .LITERAL EX1 ")"
                   .NODE(PARSER-TREE #3 #2 #1) /
       ".CHART" "(" .LIST(PARSER-CHART
                          $<1:?>(ID .USE(COMPONENT) .LITERAL
                                 ID .USE(COMPONENT) .LITERAL)
                          EX1) ")" /
       "[" "[" EX1 "]" EX1 "]" .NODE(PARSER-ERRORBLOCK #2 #1) /
       ".DEF" "(" ID ")" .NODE(PARSER-DECLARE-DEF *) /
       ".USE" "(" ID ")" .NODE(PARSER-DECLARE-USE *) /
       ".RESOLVE" "(" ID ")" .NODE(PARSER-DECLARE-RESOLVE *) /
       ".RETRACT" "(" ID ")" .NODE(PARSER-DECLARE-RETRACT *) /
       ".ASSUME"  "(" ID ")" .NODE(PARSER-DECLARE-ASSUME  *) /
       ".CONTEXT-PUSH" "(" ID ")" .NODE(PARSER-DECLARE-PUSH *) /
       ".CONTEXT-POP"  "(" ID ")" .NODE(PARSER-DECLARE-POP  *) /
       ".ERROR" .NODE(PARSER-ERROR) /
       ".FAIL"  .NODE(PARSER-FAIL) /
       ".MSG" "(" .LIST(PROGN $MINFO .NODE(AND T)) ")" /
       ".LIST" "(" ID .LITERAL EX1 ")" .NODE(PARSER-LIST #2 #1) /
       ".SEXPN" "(" EX1 ")" .NODE(PARSER-SEXPN #1) /
       ".EXECUTE" .NODE(PARSER-EXECUTE) ;

NLIST = .LIST(NCONC (ID .USE(COMPONENT)
                        .NODE(PARSER-NODE-NAME *) $NINFO /
                     .EMPTY $<1:?>NINFO)) ;

NINFO = ID .NODE(PARSER-NODE-NAME *) /
        NUMBER .NODE(PARSER-NODE-NAME *) /
        "*" .NODE(PARSER-NODE-STAR) /
        "#" NUMBER .NODE(PARSER-NODE-SHARP *) /
        "(" NLIST ")" .NODE(LIST #1) ;

MINFO = STRING .NODE(PRINAC (QUOTE *)) /
        "*"    .NODE(PRINAC (PARSER-TOKEN-MAKE)) /
        ".CR"  .NODE(TERPRI) /
        ".COL" "(" NUMBER ")" .NODE(TAB *) ;

TEX1 = .LIST(OR TEX2 $("/" TEX2)) ;

TEX2 = .LIST(AND $TEX3) ;

TEX3 = ID .USE(RULE) .NODE(*) /
       "(" TEX1 ")" /
       "$" RPTTR /
       ".ANY" ( "BUT" "(" CEX1 ")" .NODE(NOT #1) /
                 .EMPTY "(" CEX1 ")" )
               .NODE(AND #1 (PARSER-SCANCHR)) /
       ".EMPTY" .NODE(AND T) /
       ".TOKEN" .NODE(PARSER-TOKEN-START) /
       ".DELTOK" .NODE(PARSER-TOKEN-END) ;

RPTPR = "<" ("?" ":" ("?" ">" EX3 .NODE(PARSER-REPEAT #1 NIL NIL) /
                      NUM ">" EX3 .NODE(PARSER-REPEAT #1 #1 NIL ) ) /
             NUM ":" ("?" ">" EX3 .NODE(PARSER-REPEAT #1 NIL  #1) /
                      NUM ">" EX3 .NODE(PARSER-REPEAT #1 #1  #1 ) ) )
        EX3 .NODE(PARSER-REPEAT #1 NIL NIL) ;

RPTTR = "<" ("?" ":" ("?" ">" TEX3 .NODE(PARSER-REPEAT #1 NIL NIL) /
                      NUM ">" TEX3 .NODE(PARSER-REPEAT #1 #1 NIL ) ) /
             NUM ":" ("?" ">" TEX3 .NODE(PARSER-REPEAT #1 NIL  #1) /
                      NUM ">" TEX3 .NODE(PARSER-REPEAT #1 #1  #1 ) ) )
        TEX3 .NODE(PARSER-REPEAT #1 NIL NIL) ;

NUM = NUMBER .LITERAL ;

CEX1 = .LIST(OR CEX2 $("!" CEX2)) ;

CEX2 = CEX3 (":" CEX3 .NODE(AND (GE CHR #2)(LE CHR #1)) /
             .EMPTY .NODE(EQUAL CHR #1)) ;

CEX3 = NUMBER .LITERAL / "'" .LITCHAR ;
```

```
PREFIX : SPACING ;

ID : SPACING .TOKEN ALPHA $<?:80>(ALPHA / DIGIT / .ANY('_!'-!'?)) .DEL
STRING : SPACING .ANY('") .TOKEN $.ANYBUT('") .DELTOK .ANY('") ;
NUMBER : SPACING .TOKEN DIGIT $<?:2>DIGIT .DELTOK ;
CMNTCHRS : .TOKEN $.ANYBUT(']) .DELTOK ;
ALPHA : .ANY('A:'Z ! 'a:'z) ;
DIGIT : .ANY('0:'9) ;
SPACING : $.ANY(32!10!13!9) ;

.END
```

APPENDIX III
THE DEFINITION OF A PRETTYPRINTER DESCRIPTION

This appendix presents the external and internal description of a prettyprinter definition (see PPGEN). This description is from the file PPGEN.DEF. Any error in the construction of a prettyprinter using PPGEN refers to a rule in this file.

III.1. The File PPGEN.DEF

```
.DEFINE PPSYN

[ PPGEN Prettyprinter Generation
  James Neighbors -- Last Modified June 24, 1983 ]

PPSYN = ".PRETTY" "PRINTER" ID
                  .LIST(PROGN $(PMDEF / COMMENT))
        ".END" .RESOLVE(COMPONENT) ;

COMMENT = "[" CMNTCHRS "]" .NODE(NILL *) ;

PMDEF = [[ ID .DEF(COMPONENT) .LITERAL
           .MSG(.CR "Rule " * .COL(30)) "="
           PMDEF1 ";"
           .NODE(DEFINE-PP-MACRO #2 (LAMBDA (E POS)
                      (PROG (TPOS)
                            (SETQ E (CDR E))
                            (SETQ TPOS POS) #1)))]
         LINTOK .NODE(NILL) ] ;
LINTOK : .TOKEN $.ANYBUT(';) .ANY(';) .DELTOK ;

PMDEF1 = .LIST(PROGN $PPOP) ;

PPOP = STRING .NODE(PRINAC (QUOTE *) TPOS) /
       NUMBER .NODE(TYO *) /
       ".COL" "(" NUMBER .NODE(TAB *) ")" /
       ".SLM" .NODE(TAB TPOS)
          ( "(" NUMBER ")" .NODE(AND (GT (CHRPOS) *) #1) /
            .EMPTY) /
       ".LM"
          ( "(" ( "-" NUMBER .NODE(DIFFERENCE POS *) /
                  "+" NUMBER .NODE(PLUS POS *) /
                  NUMBER .NODE(PLUS POS *) ) ")" /
            .EMPTY .NODE(CHRPOS) ) .NODE(SETQ TPOS #1) /
       "#" NUMBER .NODE(PP-PRINT1 (CAR (NTH E *)) TPOS) /
       ".TREEPRINT" "(" ID .LITERAL ","
                        NUMBER .NODE(CAR (NTH E *)) ","
                        PMDEF1 "," PMDEF1 ")"
           .NODE(PP-PRINT-TREE (QUOTE #4)#3(QUOTE #2)(QUOTE #1)
                              (CONS POS TPOS)) /
       ".CHARTPRINT" "("
           .LIST(PP-PRINT-CHART .NODE(CHRPOS)
                    $<1:?>( ID     .NODE(QUOTE *)       ","
                            NUMBER .NODE(CAR (NTH E *)) ","
                            PMDEF1 .NODE(QUOTE #1)      ","
                            PMDEF1 .NODE(QUOTE #1)     (","/.EMPTY)
                                                    ))   ")" /
       ".LISTPRINT" "(" PMDEF1 ")"
           .NODE(MAP (FUNCTION (LAMBDA (TPTR)
                                 (PP-PRINT1 (CAR TPTR) TPOS)
                                 (AND (CDR TPTR) #1)))
                     E) /
        ".CHARPRINT" "(" NUMBER ")"
             .NODE(TYO (OR (INUMP (CAR (NTH E *))) 7)) ;

PREFIX : SPACING ;
ID : SPACING .TOKEN (ALPHA / .ANY('<!'*))
                    $<?:40>(ALPHA / DIGIT)
                    (.ANY('>!'*) / .EMPTY)
              .DELTOK ;
STRING : SPACING .ANY('") .TOKEN $.ANYBUT('") .DELTOK .ANY('") ;
NUMBER : SPACING .TOKEN DIGIT $<?:2>DIGIT .DELTOK ;
CMNTCHRS : .TOKEN $.ANYBUT(']) .DELTOK ;
ALPHA : .ANY('A:'Z ! 'a:'z ! '- ! '_ ! '?) ;
DIGIT : .ANY('0:'9) ;
SPACING : $.ANY(32!10!13!9) ;

.END
```

APPENDIX IV
AN EXAMPLE PRETTYPRINTER DESCRIPTION

This appendix presents an example description of a prettyprinter for the language defined in Appendix I. This description is contained in the file SIMAL.PPD and would be given to PPGEN to construct a prettyprinter. The example SIMAL program in Appendix I was printed with the prettyprinter resulting from this description.

IV.1. A SIMAL Prettyprinter Description

```
.PRETTYPRINTER SIMAL
[ SIMAL Example Language Prettyprinter ]
[ James Neighbors -- Last Modified March 11, 1982 ]

<ZIDOPS> = #1 "<ZIDOPS>" #2 ;
<OIDOPS> = #1 "<OIDOPS>" #2 ;
<OP>     = #1 "<OP>" #2 ;
ABS      = "ABS(" #1 ")" ;
ADD      = #1 "+" #2 ;
AND      = #1 "&" #2 ;
AP       = "(" .LM .TREEPRINT(SPSEQ,1,",",")") ;
APSEQ    = #1 .TREEPRINT(APSEQ,2,",",")") ;
ASELECT  = #1 "[" #2 "]" ;
ASSIGN   = .SLM #1 ":=" #2 ;
BLK      = .TREEPRINT(BLKSEQ,2,";"," ]]") ;
BLKSEQ   = #1 .TREEPRINT(BLKSEQ,2,";"," ]]") ;
BLOCK    = .SLM "[[ " .LM #1 #2 ;
DIV      = #1 "/" #2 ;
EQUAL    = #1 "=" #2 ;
EXP      = #1 "^" #2 ;
FALSE    = "FALSE" ;
FNCALL   = #1 #2 ;
FNDEF    = "$" #1 #2 .SLM .LM(2) #3 .SLM "$" ;
FOR      = .SLM "FOR " .LM #1 ":=" #2 " STEP " #3
     " TO " #4 " DO " .SLM(22) #5 ;
FOR1     = .SLM "FOR " .LM #1 ":=" #2
     " TO " #3 " DO " .SLM(22) #4 ;
FP       = "(" .LM .TREEPRINT(FPSEQ,2,",",")") ;
FPSEQ    = #1 .TREEPRINT(FPSEQ,2,",",")") ;
GOTO     = .SLM "GOTO " #1 ;
GTR      = #1 ">" #2 ;
GTREQ    = #1 ">=" #2 ;
IDIV     = #1 "//" #2 ;
IF       = .SLM "IF " #1 .LM " THEN " #2 ;
IFELSE   = .SLM "IF " #1 .LM " THEN " #2 .SLM(22) " ELSE " #3 ;
INT      = "INT(" #1 ")" ;
LABEL    = .LM(-10) .SLM #1 ":" .LM(0) #2 ;
LESS     = #1 "<" #2 ;
LESSEQ   = #1 "<=" #2 ;
LOC      = "LOCAL " .TREEPRINT(LOCSEQ,1,",",";") ;
LOFSEQ   = #1 .TREEPRINT(LOCSEQ,2,",",";") ;
MINUS    = "-" #1 ;
MPY      = #1 "*" #2 ;
NOLOC    = ;
NOT      = "%" #1 ;
NOTEQ    = #1 "#" #2 ;
```

```
NUMBER    = #1 ;
OR        = #1 "!" #2 ;
PAREN     = "(" #1 ")" ;
PGM       = .SLM ".PROGRAM " .LM
      .TREEPRINT(PGMSEQ,1,.SLM,.LM(0) .SLM ".END" .SLM) ;
PGMSEQ    = #1 .TREEPRINT(PGMSEQ,2,.SLM,.SLM ".END" .SLM) ;
PRINT     = .SLM "PRINT(" .LM .TREEPRINT(PRSEQ,1,",",",")") ;
PRSEQ     = #1 .TREEPRINT(PRSEQ,2,",",",")") ;
PROCCALL = .SLM #1 #2 ;
READCHAR = "READCHAR" ;
READNUM  = "READNUM" ;
READSTRING = "READSTRING" ;
REPEAT    = .SLM "REPEAT " .LM #1 .SLM "UNTIL " #2 ;
RETURN    = .SLM "RETURN" ;
RETURN    = .SLM "RETURN " .LM #1 ;
SASSIGN   = .SLM #1 ":=" #2 ;
SL        = "[" .LM .TREEPRINT(SLSEQ,1,",",",","]") ;
SLSEQ     = #1 .TREEPRINT(SLSEQ,2,",",",","]") ;
SQRT      = "SQRT(" #1 ")" ;
SSELECT   = #1 #2 ;
STRING    = 34 #1 34 ;
SUB       = #1 "-" #2 ;
TRUE      = "TRUE" ;
WHILE     = .SLM "WHILE " #1 .LM " DO " #2 ;
WRITECHAR = .SLM "WRITECHAR(" #1 ")" ;
WRITENUM  = .SLM "WRITENUM(" #1 ")" ;
WRITESTRING = .SLM "WRITESTRING(" #1 ")" ;

.END
```

APPENDIX V
AN EXAMPLE SET OF TRANSFORMATIONS

This appendix presents a sample set of transformations for a slightly modified version of SIMAL and its prettyprinter. The language has been modified to put objects it knows are constants into a internal form node with prefix keyword LCONST. The prettyprinter has been modified to show these constants in *{}* brackets and to print all the classes in a sensible form. Thus {?Y} represents a "match anything known to be constant". This catalog represents most of the source-to-source program transformations found in the Irvine Program Transformation Catalogue [Standish76]. The listing is in the standard XFMGEN catalog format.

V.1. SIMAL Transformations

NOTE 1997: Some past word processor (Scribe?) treated SIMAL % nots as comments so these transformations (e.g., NOT*) are truncated here.

```
5/3/79   19:18:18   SIMAL.TLB
<BOP> = {ASSIGN,EXP,DIV,IDIV,MPY,SUB,ADD,
         NOTEQ,EQUAL,GTR,LESS,GTREQ,LESSEQ,AND,OR}
<CALL> = {FNCALL,PROCCALL}
<DIV> = {DIV,IDIV}
<GE> = {GTR,GTREQ}
<LE> = {LESS,LESSEQ}
<REL> = {NOTEQ,EQUAL,GTR,LESS,GTREQ,LESSEQ}
<SEL> = {ASELECT,SSELECT}
<UOP> = {NOT,MINUS}
<BOP>CC: 12  {?X}<bop>{?Y}  =>  {?X<bop>?Y}
<BOP>EMPX: 12  *EMPTY*<bop>?X  =>  *UNDEFINED*
<BOP>IFELSEX: 4  (IF ?P THEN ?S1
                      ELSE ?S2)<bop>?X  =>  (IF ?P THEN
(?S1)<bop>?X
                                                        ELSE
(?S2)<bop>?X)
<BOP>IFX: 4  (IF ?P THEN ?S1)<bop>?X  =>  (IF ?P THEN (?S1)<bop>?X)
<BOP>UNX: 12  ?X<bop>*UNDEFINED*  =>  *UNDEFINED*
<BOP>XEMP: 12  ?X<bop>*EMPTY*  =>  *UNDEFINED*
<BOP>XIF: 3  ?X<bop>(IF ?P THEN ?S1)  =>  (IF ?P THEN ?X<bop>(?S1))
<BOP>XIFELSE: 3  ?X<bop>(IF ?P THEN ?S1)  =>  (IF ?P THEN
?X<bop>(?S1))
<BOP>XUN: 12  *UNDEFINED<bop>?X  =>  *UNDEFINED*
<DIV>0X: 9  0<div>?X  =>  0
<DIV>AMB: 10  ?A<div>-?B  =>  -(?A<div>?B)
<DIV>MAB: 10  -?A<div>?B  =>  -(?A<div>?B)
```

```
<DIV>MAMB: 12  -?A<div>-?B  =>  ?A<div>?B
<DIV>X0: 12  ?X<div>0  =>  *UNDEFINED*
<DIV>X1: 12  ?X<div>1  =>  ?X
<DIV>XX: 11  ?X<div>?X  =>  1
<REL>0S: 10  0<rel>?A-?B  =>  ?A<rel>?B
<REL>1D: 9  1<rel>?A/?B  =>  (IF ?B>0 THEN ?B<rel>?A
                                      ELSE ?A<rel>?B)
<REL>AA: 10  ?A+{?C}<rel>?B+{?C}  =>  ?A<rel>?B
<REL>DD: 9  ?A/{?C}<rel>?B/{?C}  =>  (IF {?C}>0 THEN ?A<rel>?B
                                                ELSE ?B<rel>?A)
<REL>MM: 9  ?A*{?C}<rel>?B*{?C}  =>  (IF {?C}>0 THEN ?A<rel>?B
                                                ELSE ?B<rel>?A)
<REL>S0: 10  ?A-?B<rel>0  =>  ?A<rel>?B
<REL>SS: 10  ?A-{?C}<rel>?B-{?C}  =>  ?A<rel>?B
<UOP>C: 12  <uop>{?X}  =>  {<uop>?X}
<UOP>EMP: 12  <uop>*EMPTY*  =>  *UNDEFINED*
<UOP>IF: 4  <uop>(IF ?P THEN ?S1)  =>  (IF ?P THEN <uop>(?S1))
<UOP>IFELSE: 3  <uop>(IF ?P THEN ?S1
                            ELSE ?S2)  =>  (IF ?P THEN <uop>(?S1)
                                                  ELSE <uop>(?S2))
<UOP>UN: 12  <uop>*UNDEFINED*  =>  *UNDEFINED*
ADD0X: 12  0+?X  =>  ?X
ADDAMB: 10  ?A+-?B  =>  ?A-?B
ADDDD: 5  ?A/?B+?C/?D  =>  (?A*?D+?B*?C)/(?B*?D)
ADDDX: 3  ?A/?B+?C  =>  (?A+?C*?B)/?B
ADDMAB: 9  -?A+?B  =>  ?B-?A
ADDMAMB: 12  -?A+-?B  =>  -(?A+?B)
ADDX0: 12  ?X+0  =>  ?X
ADDXD: 3  ?A+?B/?C  =>  (?A*?C+?B)/?C
ANDFX: 11  FALSE&?X  =>  FALSE
ANDNOTXX: 11  ANDOO: 11  (?X!ANDTX: 12  TRUE&?X  =>  ?X
ANDXF: 11  ?X&FALSE  =>  FALSE
ANDXNOTX: 11  ?X&ANDXOR: 9  ?X&(?X!?Y)  =>  ?X
ANDXT: 12  ?X&TRUE  =>  ?X
ANDXX: 11  ?X&?X  =>  ?X
ANDXY: 3  ?X&?Y  =>  (IF ?X THEN ?Y
                            ELSE FALSE)
ASSIGNID: 11  ?X:=?X  =>  *EMPTY*
ASSIGNXX: 11  ?X:=?Y;
              ?X:=?Z  =>  ?X:=?Z
BLOCKBLOCKN: 12  [[LOCAL ?X;
                   [[?S]]]]  =>  [[LOCAL ?X;
                                   ?S]]
BLOCKEMP: 12  [[LOCAL ?X;
                *EMPTY*]]  =>  *EMPTY*
BLOCKN<CALL>: 12  [[?X(?Y)]]  =>  ?X(?Y)
BLOCKNASSIGN: 12  [[?X:=?Y]]  =>  (?X:=?Y)
BLOCKNBLOCKN: 12  [[[[?S]]]]  =>  [[?S]]
BLOCKNEMP: 12  [[*EMPTY*]]  =>  *EMPTY*
BLOCKNFOR: 12  [[FOR ?V:=?W STEP ?X TO ?Y DO
                     ?Z]]  =>  (FOR ?V:=?W STEP ?X TO ?Y DO
                                    ?Z)
BLOCKNIF: 12  [[IF ?P THEN ?S1]]  =>  (IF ?P THEN ?S1)
BLOCKNIFELSE: 12  [[IF ?P THEN ?S1
                          ELSE ?S2]]  =>  (IF ?P THEN ?S1
                                                 ELSE ?S2)
BLOCKNREPEAT: 12  [[REPEAT ?X
                           UNTIL ?Y]]  =>  (REPEAT ?X
                                                   UNTIL ?Y)
BLOCKNWHILE: 12  [[WHILE ?X DO ?Y]]  =>  (WHILE ?X DO ?Y)
DIVDD: 5  (?A/?B)/(?C/?D)  =>  (?A*?D)/(?B*?C)
DIVDX: 3  (?A/?B)/?C  =>  ?A/(?B*?C)
DIVXD: 3  ?C/(?A/?B)  =>  (?B*?C)/?A
EQUALMAMB: 12  -?A=-?B  =>  ?A=?B
EQUALXX: 11  ?X=?X  =>  TRUE
EXP00: 12  0^0  =>  *UNDEFINED*
EXP1X: 14  1^?X  =>  1
EXPAMB: 10  ?A^-?B  =>  (1/?A^?B)
EXPX0: 9  ?X^0  =>  1
EXPX1: 14  ?X^1  =>  ?X
EXPX2: 9  ?X^2  =>  ?X*?X
FNCALL: 12  LOG(0)  =>  *UNDEFINED*
FOREMP: 11  FOR ?W:=?X STEP ?Y TO ?Z DO
                *EMPTY*  =>  *EMPTY*
FORFUNROLL: 1  FOR ?V:=?W STEP ?X TO ?Y DO
                   ?Z(?V)  =>  [[IF ?W<=?Y THEN ?Z(?V:=?W);
                                 FOR ?V:=?W+?X STEP ?X TO ?Y DO
```

```
                                     ?Z(?V)]]
FORREDUCE: 5  FOR ?V:=?W STEP ?X TO ?Y DO
                  ?Z(?V*?Q)  =>  FOR ?V:=?W*?Q STEP ?X*?Q TO ?Y*?Q
DO
                                     ?Z(?V)
FORTOWHILE: 2  FOR ?V:=?W STEP ?X TO ?Y DO
                   ?Z  =>  [[?V:=?W;
                             WHILE ?V-?Y*SIGN(?X)<=0 DO [[?Z;

?V:=?V+?X]]]]
FORUN: 12  FOR ?V:=?W STEP ?X TO ?Y DO
               *UNDEFINED*  =>  *UNDEFINED*
FORXX: 11  FOR ?W:=?X STEP ?Y TO ?X DO
               ?Z  =>  [[?W:=?X;
                         ?Z]]
GTREQXX: 11  ?X>=?X  =>  TRUE
GTREQXY: 10  ?X>=?Y  =>
GTRMAMB: 12  -?A>-?B  =>  ?A<?B
GTRXX: 11  ?X>?X  =>  FALSE
GTRXY: 10  ?X>?Y  =>  IFELSE<CALL>: 6  IF ?P THEN ?Y(?S1)
                       ELSE ?Y(?S2)  =>  ?Y(IF ?P THEN ?S1
                                                ELSE ?S2)
IFELSE<SEL>: 6  IF ?P THEN ?Y[?S1]
                      ELSE ?Y[?S2]  =>  ?Y[IF ?P THEN ?S1
                                               ELSE ?S2]
IFELSEEMPX: 12  IF ?P THEN *EMPTY*
                      ELSE ?X  =>  IF %?P THEN ?X
IFELSEF: 12  IF FALSE THEN ?S1
                      ELSE ?S2  =>  ?S2
IFELSEIFXIFX: 6  IF ?P THEN IF ?X THEN ?W
                       ELSE IF ?Y THEN ?W  =>  IF IF ?P THEN ?X
                                                     ELSE ?Y THEN
?W
IFELSENOT: 12  IF %?P THEN ?S1 ELSE ?S2  =>  IF ?P THEN ?S2
                                                  ELSE ?S1
IFELSET: 12  IF TRUE THEN ?S1
                     ELSE ?S2  =>  ?S1
IFELSEUNX: 10  IF ?P THEN *UNDEFINED*
                     ELSE ?X  =>  *UNDEFINED*
IFELSEXEMP: 12  IF ?P THEN ?S1
                      ELSE *EMPTY*  =>  IF ?P THEN ?S1
IFELSEXFT: 12  IF ?X THEN FALSE
                     ELSE TRUE  =>  IFELSEXTF: 12  IF ?X THEN TRUE
                     ELSE FALSE  =>  ?X
IFELSEXUN: 10  IF ?P THEN ?X
                     ELSE *UNDEFINED*  =>  *UNDEFINED*
IFELSEXX: 11  IF ?P THEN ?S1
                    ELSE ?S1  =>  ?S1
IFEMP: 11  IF ?P THEN *EMPTY*  =>  *EMPTY*
IFF: 12  IF FALSE THEN ?S1  =>  *EMPTY*
IFIF: 11  IF ?X THEN IF ?Y THEN ?S1  =>  IF ?X&?Y THEN ?S1
IFLESE2IFELSE: 11  IF ?X THEN IF ?Y THEN ?S1
                                  ELSE ?S2
                         ELSE ?S2  =>  IF ?X&?Y THEN ?S1
                                           ELSE ?S2
IFLESSEQFOR: 11  IF ?X<=?Y THEN FOR ?W:=?X STEP ?Z TO ?Y DO
                                  ?S1  =>  FOR ?W:=?X STEP ?Z TO
?Y DO
                                               ?S1
IFLESSFOR: 11  IF ?X<?Y THEN FOR ?W:=?X STEP ?Z TO ?Y DO
                               ?S1  =>  FOR ?W:=?X STEP ?Z TO ?Y
DO
                                            ?S1
IFT: 12  IF TRUE THEN ?S1  =>  ?S1
IFUN: 10  IF ?P THEN *UNDEFINED*  =>  *UNDEFINED*
LABELIFX: 10  ?X:
                IF ?P THEN [[?S;
                             GOTO ?X]]  =>  ?X:
                                            WHILE ?P DO ?S
LESSEQMAMB: 12  -?A<=-?B  =>  ?A>=?B
LESSEQXX: 11  ?X<=?X  =>  TRUE
LESSMAMB: 12  -?A<-?B  =>  ?A>?B
LESSXX: 11  ?X<?X  =>  FALSE
MINUS0: 14  -0  =>  0
MINUSMINUSX: 12  --?X  =>  ?X
MINUSSUBAMB: 9  -(?A--?B)  =>  (?B-?A)
MPY0X: 11  0*?X  =>  0
```

```
MPY1X: 12  1*?X  =>  ?X
MPYAMB: 10  ?A*-?B  =>  -(?A*?B)
MPYDD: 5  (?A<div>?B)*(?C<div>?D)  =>  (?A*?C)<div>(?B*?D)
MPYDX: 3  (?A/?B)*?C  =>  (?A*?C)/?B
MPYMAB: 10  -?A*?B  =>  -(?A*?B)
MPYMAMB: 12  -?A*-?B  =>  ?A*?B
MPYX0: 11  ?X*0  =>  0
MPYX1: 12  ?X*1  =>  ?X
MPYXD: 3  ?C*(?A/?B)  =>  (?A*?C)/?B
NOTEQMAMB: 12  -?A#-?B  =>  ?A#?B
NOTEQUAL: 8
NOTEQXX: 11  ?X#?X  =>  FALSE
NOTEQXY: 10  ?X#?Y  =>
NOTF: 12
NOTGTR: 12
NOTGTREQ: 12
NOTLESS: 8
NOTLESSEQ: 8
NOTNOT: 12
NOTNOTEQ: 12
NOTT: 12
NOTX: 3
                  ELSE TRUE)
OR<GE>EQ: 9  ?A<ge>?B!?A=?B  =>  ?A>=?B
OR<LE>EQ: 9  ?A<le>?B!?A=?B  =>  ?A<=?B
ORAA: 11  (?X&OREQ<GE>: 9  ?A=?B!?A<ge>?B  =>  ?A>=?B
OREQ<LE>: 9  ?A=?B!?A<le>?B  =>  ?A<=?B
ORFX: 12  FALSE!?X  =>  ?X
ORNOTXX: 11  ORTX: 11  TRUE!?X  =>  TRUE
ORXAND: 9  ?X!(?X&?Y)  =>  ?X
ORXF: 12  ?X!FALSE  =>  ?X
ORXNOTX: 11  ?X!ORXT: 11  ?X!TRUE  =>  TRUE
ORXX: 11  ?X!?X  =>  ?X
ORXY: 3  ?X!?Y  =>  (IF ?X THEN TRUE
                           ELSE ?Y)
PARCONST: 12  ({?X})  =>  {?X}
PAREMP: 12  (*EMPTY*)  =>  *EMPTY*
PARF: 12  (FALSE)  =>  FALSE
PARPAR: 12  ((?X))  =>  (?X)
PART: 12  (TRUE)  =>  TRUE
PARUN: 12  (*UNDEFINED*)  =>  *UNDEFINED
REPEATEMP: 9  REPEAT *EMPTY*
                     UNTIL ?P  =>  *EMPTY*
REPEATIFELSE: 1  REPEAT IF ?Q THEN ?R
                                 ELSE ?S
                        UNTIL ?P  =>  REPEAT [[WHILE ?Q DO ?R;
                                                  ?S]]
                                                UNTIL ?P
REPEATSUN: 10  REPEAT ?S
                      UNTIL *UNDEFINED*  =>  *UNDEFINED*
REPEATUNP: 12  REPEAT *UNDEFINED*
                      UNTIL ?P  =>  *UNDEFINED*
SEMICAW: 2  ?X:=?Y(?X);
            WHILE ?P(?X) DO [[?Q(?X);
                   ?X:=?Y(?X)]]  =>  WHILE ?P(?X:=?Y(?X)) DO ?Q(?X)
SEMICBLOCKN: 12  [[?S1]];
                 ?S2  =>  ?S1;
                          ?S2
SEMICEMPX: 12  *EMPTY*;
               ?X  =>  ?X
SEMICIFELSEX: 2  IF ?P THEN ?X
                       ELSE ?Y;
                 ?S1  =>  IF ?P THEN [[?X;
                                       ?S1]]
                            ELSE [[?Y;
                                       ?S1]]
SEMICIFIF: 9  IF ?P THEN ?X;
              IF                                         ELSE ?Y
SEMICLEMPS: 12  ?X:
                  *EMPTY*;
                ?S  =>  ?X:
                          ?S
SEMICLXIF: 10  ?X:
                 ?S;
               IF                                           REPEAT
?S
                                                UNTIL ?Y
```

```
SEMICXEMP: 12  ?X;
               *EMPTY*  =>  ?X
SEMICXIFELSE: 1  ?S1;
                 IF ?P THEN ?X
                       ELSE ?Y  =>  IF ?P THEN [[?S1;
                                                 ?X]]
                                          ELSE [[?S1;
                                                 ?Y]]
SEMICXWHILEX: 6  ?S;
                 WHILE
UNTIL ?X
SUB0X: 12  0-?X  =>  -(?X)
SUBDD: 5  ?A/?B-?C/?D  =>  (?A*?D-?B*?C)/(?B*?D)
SUBDX: 3  ?A/?B-?C  =>  (?A-?C*?B)/?B
SUBMAB: 10  -?A-?B  =>  -(?A+?B)
SUBMAMB: 11  -?A--?B  =>  ?B-?A
SUBX0: 12  ?X-0  =>  ?X
SUBXD: 3  ?A-?B/?C  =>  (?A*?C-?B)/?C
SUBXX: 11  ?X-?X  =>  0
WHILEEMP: 9  WHILE ?P DO *EMPTY*  =>  *EMPTY*
WHILEF: 12  WHILE FALSE DO ?S  =>  *EMPTY*
WHILEIFELSE: 1  WHILE ?P DO IF ?Q THEN ?R
                            ELSE ?S  =>  WHILE ?P DO [[WHILE ?Q DO
?R;
                                                              ?S]]
WHILEPUN: 10  WHILE ?P DO *UNDEFINED*  =>  *UNDEFINED*
WHILEUNS: 12  WHILE *UNDEFINED* DO ?S  =>  *UNDEFINED*
```

APPENDIX VI
THE DEFINITION OF A COMPONENT INSERTION FILE

The definition of a packet of components to be added to a refinement library is described in this appendix. Errors in scanning these packets in REFGEN refer to the file REFGEN.DEF given below.

VI.1. The File REFGEN.DEF

```
.DEFINE COMPONENTS
[ Component Library Scanner ]
[ James Neighbors -- Last Modified December 29, 1982 ]

COMPONENTS = .LIST(COMSET $<1:?>COMPONENT) EOF .RESOLVE(COMPONENT) ;

COMPONENT = "COMPONENT:"
            .LIST(COMLIST
                  NAME .MSG(.CR "Component " * .COL(30))
                       .DEF(COMPONENT)
                       .NODE(COMPONENT #1)
                  ("(" .LIST(CPARAMS CPNAM $("," CPNAM)) ")" /
                   .EMPTY .NODE(CPARAMS)) CR
                  $( "PURPOSE:" MLTEXT .NODE(PURPOSE #1) /
                     "IOSPEC:" MLTEXT .NODE(IOSPEC #1) /
                     "DECISION:" MLTEXT .NODE(DECISION #1) )
                  $BLINE
                  .LIST(REFSET $<1:?>REFMNT))
            "END" "COMPONENT" $<1:?>BLINE ;

CPNAM = "'" NAME .NODE(CPQUOTE #1) / NAME ;

REFMNT ="REFINEMENT:"
        .LIST(REFLIST
              REFNAME .MSG(.CR "   Refinement " * .COL(30))
                      .NODE(REFINEMENT #1) CR
               $("BACKGROUND:" MLTEXT .NODE(BACKGROUND #1) /
                 "INSTANTIATION:"
                     .LIST(INSTANTIATION NAME $("," NAME))
                     CR /
                 "ASSERTIONS:" ASSERTIONSET /
                 "CONDITIONS:" CONDITIONSET /
                 "RESOURCES:" MLTEXT .NODE(RESOURCES #1) /
                 "ADJUSTMENTS:" MLTEXT .NODE(ADJUSTMENTS #1) /
                 "GLOBALS:" .LIST(GLOBALS NAME $("," NAME)) CR /
                 "LABELS:" .LIST(LABELS NAME $("," NAME)) CR )
              ("CODE:" NAME "." NAME CR
                 .NODE(PARSE-DOMAIN #2 #1) .EXECUTE LINE
                 .NODE(CODE #3) .NODE(DOMAIN #2) /
               "INTERNAL:" NAME CR
                 ILIST LINE
                 .NODE(PARSE-INTERNAL #2 #1) .EXECUTE
                 .NODE(CODE #3) .NODE(DOMAIN #2) /
              "DIRECTIVE:"
                  ("FUNCTION" ("DEFINITION" .NODE(DIRECTIVE DIRECTIVE-
                               "CALL" .NODE(DIRECTIVE DIRECTIVE-FN-CAL
                   "DEFER" .NODE(DIRECTIVE DIRECTIVE-DEFER) )
                 CR ))
          "END" "REFINEMENT" $<1:?>BLINE ;

CONDITIONSET = .LIST(CONDITIONS CONDITION $(WHITE CONDITION)) ;
ASSERTIONSET = .LIST(ASSERTIONS ASSERTION $(WHITE ASSERTION)) ;
CONDITION = NAME NAME "as" NAME .NODE(CONDITION #3 #2 #1) CR ;
ASSERTION = NAME NAME "as" NAME .NODE(ASSERTION #3 #2 #1) CR ;

ILIST = NAME / "(" .SEXPN(NAME $<?:?>(ILIST/CR)) ")" ;
REFNAME = REFNTOK .LITERAL ;
NAME = NAMETOK .LITERAL ;
MLTEXT = MLTOK .LITERAL ;

PREFIX : $.ANY(32!9) ;
WHITE : $<1:?>.ANY(32!9) ;
CR : PREFIX .ANY(13!10) $.ANY(13!10) ;
EOF : .ANY(26) ;
REFNTOK : PREFIX .TOKEN $<1:?>(NAMECHR/.ANY(32)) .DELTOK ;
NAMETOK : PREFIX .TOKEN $<1:?>NAMECHR .DELTOK ;
NAMECHR : .ANY('A:'Z!'a:'z!'0:'9!'*!'-!'_) ;
MLTOK : PREFIX .TOKEN LINE $(.ANY(32!9) LINE) .DELTOK ;
LINE : $.ANY(32:125!9) CR ;
BLINE : $.ANY(32) CR ;

.END
```

APPENDIX VII
THE DEFINITION OF TACTICS

This is the definition of the interpreter for the TACTICS subsystem. Any errors in the TACTICS subsystem refer to the file TACTICS.DEF which is reproduced below. Any error in the construction of a prettyprinter using PPGEN refers to a rule in this file.

VII.1. The File TACTIC.DEF

```
.DEFINE TACTCMD
[ Tactics Parser Uses .EXECUTE to be an Interpreter ]
[ James Neighbors  Last Modified -- December 12, 1982 ]
[   NOTE: terrible use of TACTIC-*-KLUGE should be removed! ]

TACTCMD = $( (TACDEF / TACLIS / TACDEL /
              "LOAD" NAME  .NODE(TACTIC-LOAD #1) /
              "HELP"       .NODE(TACTIC-HELP) )
            ";"
            .EXECUTE )
           "EXIT" ;

TACDEF = "DEFINE" (NAME ("." NAME "=" RULE .NODE(TACTIC-DEFINE #3 #2 #
                         .EMPTY "="   RULE .NODE(TACTIC-DEFINE #2 (TAC
                   .EMPTY "="         RULE .NODE(TACTIC-DEFINE (TACTIC
TACDEL = "DELETE" (NAME ("." NAME .NODE(TACTIC-DELETE #2 #1) /
                         .EMPTY   .NODE(TACTIC-DELETE #1 (TACTIC-BLANK
                   .EMPTY         .NODE(TACTIC-DELETE (TACTIC-BLANK-KL
TACLIS = "LIST" (NAME ("." NAME (">" NAME .NODE(TACTIC-LIST #3 #2 #1)
                                 .EMPTY   .NODE(TACTIC-LIST #2 #1 (TAC
                       .EMPTY   (">" NAME .NODE(TACTIC-LIST #2 (TACTIC
                                 .EMPTY   .NODE(TACTIC-LIST #1 (TACTIC
                 .EMPTY         (">" NAME .NODE(TACTIC-LIST (TACTIC-BL
                                 .EMPTY   .NODE(TACTIC-LIST (TACTIC-BL

RULE = .LIST(PROGN RCMD $("," RCMD)) .NODE(QUOTE #1) ;
```

```
RCMD = STRING      .NODE(TACTIC-MESSAGE #1) /
       "COMPONENT" .NODE(TACTIC-RPCFIELD COMPONENT) /
       "PURPOSE"   .NODE(TACTIC-RPCFIELD PURPOSE) /
       "IOSPEC"    .NODE(TACTIC-RPCFIELD IOSPEC) /
       "DECISION"  .NODE(TACTIC-RPCFIELD DECISION) /
       "LOC" (NUMBER .NODE(TFMREF-LOC #1) /
              .EMPTY .NODE(TFMREF-LOC (TACTIC-BLANK-KLUGE)) ) /
       "USE" ("DEFAULT" (NAME   .NODE(REFINE-USE-NUM 1 #1) /
                         .EMPTY .NODE(REFINE-USE-NUM 1 (TACTIC-BLANK-K
              NUMBER    (NAME   .NODE(REFINE-USE-NUM #2 #1) /
                         .EMPTY .NODE(REFINE-USE-NUM #1 (TACTIC-BLANK-
       "TRY" ("DEFAULT" (NAME   .NODE(REFINE-TRY-NUM 1 #1) /
                         .EMPTY .NODE(REFINE-TRY-NUM 1 (TACTIC-BLANK-K
              NUMBER    (NAME   .NODE(REFINE-TRY-NUM #2 #1) /
                         .EMPTY .NODE(REFINE-TRY-NUM #1 (TACTIC-BLANK-
       "[" ("ALL"  RPRED "," RCMND .NODE(TACTIC-REFSCAN (TACTIC-ALL-KL
            NUMBER RPRED "," RCMND .NODE(TACTIC-REFSCAN #3  #2 #1) ) "
       NAME .NODE(TACTIC-CALL #1) ;

RPRED = "<" .LIST(AND SPRED $("&" SPRED)) ">" / .EMPTY .NODE(OR T) ;
RCMND = .LIST(PROGN SCMD $("," SCMD)) ;

SPRED = REFFLD ("IS" NAME .NODE(EQ #2 #1) / .EMPTY) /
        "NO" REFFLD .NODE(NOT #1) /
        "AVAILABLE" ("FUNCTION" .NODE(REFINE-FUNCTION-ALREADY?) /
                     "RESOURCE" .NODE(TACTIC-RRCHECK) ) /
        NAME "INSTANTIATION" .NODE(TACTIC-INSTANTIATION-AVAILABLE? #1)

SCMD = STRING          .NODE(TACTIC-MESSAGE #1) /
       "REFINEMENT"    .NODE(TACTIC-RPRFIELD REFINEMENT) /
       "CONDITIONS"    .NODE(TACTIC-RPRFIELD CONDITIONS) /
       "BACKGROUND"    .NODE(TACTIC-RPRFIELD BACKGROUND) /
       "DIRECTIVE"     .NODE(TACTIC-RPRFIELD DIRECTIVE) /
       "INSTANTIATION" .NODE(TACTIC-RPRFIELD INSTANTIATION) /
       "ASSERTIONS"    .NODE(TACTIC-RPRFIELD ASSERTIONS) /
       "RESOURCES"     .NODE(TACTIC-RPRFIELD RESOURCES) /
       "ADJUSTMENTS"   .NODE(TACTIC-RPRFIELD ADJUSTMENTS) /
       "DOMAIN"        .NODE(TACTIC-RPRFIELD DOMAIN) /
       "USE" (NAME .NODE(REFINE-USE #1) / .EMPTY .NODE(REFINE-USE (TAC
       "TRY" (NAME .NODE(REFINE-TRY #1) / .EMPTY .NODE(REFINE-TRY (TAC

REFFLD = "REFINEMENT"    .NODE(TACTIC-RRFIELD REFINEMENT) /
         "CONDITIONS"    .NODE(TACTIC-RRFIELD CONDITIONS) /
         "BACKGROUND"    .NODE(TACTIC-RRFIELD BACKGROUND) /
         "DIRECTIVE"     .NODE(TACTIC-RRFIELD DIRECTIVE) /
         "INSTANTIATION" .NODE(TACTIC-RRFIELD INSTANTIATION) /
         "ASSERTIONS"    .NODE(TACTIC-RRFIELD ASSERTIONS) /
         "RESOURCES"     .NODE(TACTIC-RRFIELD RESOURCES) /
         "ADJUSTMENTS"   .NODE(TACTIC-RRFIELD ADJUSTMENTS) /
         "DOMAIN"        .NODE(TACTIC-RRFIELD DOMAIN) ;

NAME = NAMTOK .NODE(INTERN (QUOTE *)) ;
STRING = STRTOK .LITERAL ;
NUMBER = NUMTOK .LITERAL ;

PREFIX : $.ANY(32!10!13!9) ;
NAMTOK : PREFIX .TOKEN .ANY('a:'z!'A:'Z!'*) $.ANY('a:'z!'A:'Z!'*!'0:'9
The File TACTIC.DEF                                                  99

STRTOK : PREFIX .ANY('") .TOKEN $.ANYBUT('") .DELTOK .ANY('") ;
NUMTOK : PREFIX .TOKEN .ANY('0:'9) $<?:5>.ANY('0:'9) .DELTOK ;

.END
```

APPENDIX VIII
DRACO TERMINAL DEFINITION

Draco can use a terminal's special features if the terminal type is defined and stored in a file named <termtype>.TRM. A new terminal type is defined as a a set of simple LISP functions that are known to Draco. The functions for a specific terminal are invoked by setting the terminal type using the SET command of DRACO_MENU.

An example for the Zenith-Heath H19 terminal in its ANSI configuration will serve as a guide.

```
{ clear the screen - sends <esc>[2J }
(DE TERM-CLEAR NIL (TERM-MSG 27. "[2J"))

{ address the cursor }
(DE TERM-CUP (LINE COL) (TERM-MSG 27. "[" LINE ";" COL "H"))

{ erase the current line }
(DE TERM-ERASE-LINE NIL (TERM-MSG 27. "[2K"))

{ initialize the terminal mode }
(DE TERM-INIT NIL (TERM-SM 1.))

{ message in inverse video }
(DE TERM-INVERSE (S)
     (TERM-MSG 27. "[7m")
     (EVAL (CONS 'TERM-MSG S))
     (TERM-MSG 27. "[0m"))

{ send a series of strings and ASCII codes to the terminal }
(DE TERM-MSG (L)
    (PROG (S)
     LOOP (SETQ S (CAR L))
     (COND [(NULL L) (RETURN)]
   [(STRINGP S)
    (MAPC (FUNCTION OUTCHR) (AEXPLODEC S))]
   [(NUMBERP S) (TYO S)]
   [(LITATOM S)
    (COND [(EQ S T) (TERPRI)]
          [(CONSP (GET S 'VALUE))
           (MAPC (FUNCTION OUTCHR)
                 (AEXPLODEC (EVAL S)))]
          [(MAPC (FUNCTION OUTCHR) (AEXPLODEC S))])]
   [(CONSP S)
    (COND [(EQ (CAR S) 'E) (EVAL (CADR S))]
          [(MAPC (FUNCTION OUTCHR)
                 (AEXPLODEC (EVAL S)))])])
     (SETQ L (CDR L))
     (GO LOOP)))

 { no access protected areas }
 (DE TERM-PRCP NIL (TERM-MSG 27. "[u"))

 { access protected areas }
 (DE TERM-PSCP NIL (TERM-MSG 27. "[s"))

 { send terminal commands }
 (DE TERM-RM (M) (TERM-MSG 27. "[>" M "l"))
 (DE TERM-SM (M) (TERM-MSG 27. "[>" M "h''))

 { display status line }
 (DF TERM-STATUS (S)
     (TERM-PSCP)
     (TERM-CUP 25. 0.)
     (TERM-ERASE-LINE)
     (EVAL (CONS 'TERM-MSG S))
     (TERM-PRCP))

{ reset terminal state on exit }
(DE TERM-TERM NIL (TERM-RM 1.))

{ clear and home before message }
(DE TERM-TITLE (S) (TERM-CLEAR) (EVAL (CONS 'TERM-MSG S)))
```

APPENDIX IX
TACTICS PRETTYPRINTER DEFINITION

The file TACTICS.PPD shown below contains the definition of the tactics prettyprinter.

```
.PRETTYPRINTER TACTICS
[ PrettyPrinter for Internal Domain of TACTICS ]
[ James Neighbors -- Last Modified August 26, 1982 ]
```

```
TACTICS = .LISTPRINT(.SLM) .SLM ;
TACTIC  = #2 ;
CMDGRP  = .LISTPRINT(.SLM) .SLM 10 13 ;
CMD     = "DEFINE " #3 "." #1 " = " .LM #2 ";" ;
PROGN   = .LISTPRINT("," .SLM(40)) ;
TACTIC-RPCFIELD = #1 ;
TFMREF-LOC     = "LOC" .LISTPRINT(" ") ;
REFINE-USE-NUM = "USE" .LISTPRINT(" ") ;
REFINE-TRY-NUM = "TRY" .LISTPRINT(" ") ;
REFINE-USE = "USE " .LISTPRINT(" ") ;
REFINE-TRY = "TRY " .LISTPRINT(" ") ;
TACTIC-CALL = #1 ;
TACTIC-MESSAGE = 34 #1 34 ;
TACTIC-BLANK-KLUGE = ;
TACTIC-ALL-KLUGE = "ALL" ;
TACTIC-REFSCAN = "[" .LM #1 #2 "," #3 "]" ;
AND     = "<" .LISTPRINT(" &" .SLM(40)) ">" ;
OR      = ;
EQ      = #1 " IS " #2 ;
TACTIC-RRFIELD = #1 ;
NOT     = "NO " #1 ;
REFINE-FUNCTION-ALREADY? = "AVAILABLE FUNCTION" ;
TACTIC-RRCHECK           = "AVAILABLE RESOURCE" ;
TACTIC-INSTANTIATION-AVAILABLE? = #1 " INSTANTIATION" ;
TACTIC-RPRFIELD = #1 ;
INTERN  = #1 ;
QUOTE   = #1 ;

.END
```

APPENDIX X
DRACO ERROR, NOTE, AND SYSERR MESSAGES

There are three basic kinds of messages from Draco: ERR:, NOTE: and SYSERR:. An ERR: is an error condition caused by a domain builder or user and is handled by Draco. A NOTE: is a message given only for the users information, no problem or extraordinary event has occured but the user's environment has in someway been modified. The creation of a file is an example of where a NOTE: is used. A SYSERR: is a disasterous error in the Draco mechanisim itself which was caught by an internal consistency check within Draco. The user should *never* save anything after a SYSERR: unless directed it is alright to do so.

```
"ERR: " NAME " not a legal insertion command"
"ERR: " NAME " name already ready in use (" NAME ")."
"ERR: " TRANSFORMATION " not a transformation"
"ERR: " TRANSFORMATION " not a transformation"
"ERR: " INSTANTIATION-TYPE " instantiation illegal"
"ERR: " COMPONENT " needs an instantiation"
"ERR: " FUNCTION " may not be EXECUTED."
"ERR: " CLASS " not a class"
"ERR: CLASS too few arguments" CLASS
"ERR: EXECUTE call in wrong form"
"ERR: Syntax error - rule " RULE
"ERR: TRANS wrong number of arguments" TRANSFORMATION
"ERR: TRANSFORM 1st arg must be less or equal to 2nd arg"
"ERR: attempt to remove a mark from the node stack"
"ERR: can't read " FILE "!"
"ERR: can't read " FILE "! - ^ or try again"
"ERR: end of file not recognized"
"ERR: enter number or OK"
"ERR: errorblock in rule " RULE " has failed to recover"
"ERR: no " GROUP " group of tactics loaded"
"ERR: no " TACTIC "." CMD " tactic loaded"
"ERR: no " DOMAIN " domain parser available."
"ERR: no component for " COMPONENT " in " DOMAIN " domain"
"ERR: no domain instance has been selected"
"ERR: no instance has been selected"
"ERR: no locale has been selected"
"ERR: no locale selected"
"ERR: no locale selected"
"ERR: no mark on stack to form list"
"ERR: no prettyprinter available"
"ERR: no refinement number " NUMBER
"ERR: no tactic " TACTIC "." CMD " to delete"
"ERR: no tactic group called " GROUP " to delete"
```

```
 "ERR: no tactics loaded"
 "ERR: no tactics to delete"
 "ERR: non numeric argument "
 "ERR: non prefix structure" LIST
 "ERR: not a single top level"
 "ERR: number too large"
 "ERR: refinement library " DOMAIN ".RLB unavailable"
 "ERR: rule " RULE " failed but constructed nodes"
 "ERR: rule " RULE " succeeded but did not return one node"
 "ERR: syntax rule " RULE
 "ERR: transformation library " DOMAIN ".TLB unavailable"
 "ERR: transformation library " FILE " unavailable for suggestions"
 "ERR: underflow of the node stack"
 "NOTE: " FILE " saved"
 "NOTE: " FILE " created"
 "NOTE: " DOMAIN " domain being removed"
 "NOTE: adding to an existing transformation library"
 "NOTE: creating a component library for the domain"
 "NOTE: creating a new transformation library"
 "NOTE: domain already selected"
 "NOTE: existing component library contains:"
 "NOTE: existing tactic " TACTIC " replaced"
 "NOTE: file " FILE " accessed from Draco disk area."
 "NOTE: file last modified on " DATE
 "NOTE: function *USER*." FUNCTION ".0 defined"
 "NOTE: function definition exhausted domain instance"
 "NOTE: insertion file components replace library components"
 "NOTE: new domain instance automatically selected"
 "NOTE: no domain instance selected"
 "NOTE: no instance of " DOMAIN " domain in locale of refinement"
 "NOTE: no prettyprinter available"
 "NOTE: no tactics have been loaded"
 "NOTE: refinement deferred by directive"
 "NOTE: refinement index loaded"
 "NOTE: refinement merged a domain"
 "NOTE: refinement replaced Draco top level"
 "NOTE: refinement replaced a domain"
 "NOTE: resource check RRCHECK is a stub"
 "NOTE: tactics interrupted by user"
 "NOTE: transformations loaded"
 "NOTE:function " DOMAIN "." COMPONENT ".0 defined"
 "SYSERR: atomic backtracking alternative"
 "SYSERR: chr pointer not on line"
 "SYSERR: failure to unlink in RFNDEF"
 "SYSERR: illegal domain internal structure"
 "SYSERR: illegal menu form"
 "SYSERR: illegal prefix internal form in PATTACH1" FORM
 "SYSERR: illegal prefix internal form in RSCAN" FORM
 "SYSERR: illegal prefix internal form in TSCAN" FORM
 "SYSERR: inline refinement with wrong number of arguments"
 "SYSERR: no binding for ?" PATVAR " in RHS of transform " TFM
 "SYSERR: no mark on stack in .LIST"
 "SYSERR: rule " PARSE-RULE " .TREE has no *MARK*"
 "SYSERR: transform should have refined"
 "SYSERR: unknown directive"
```