LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Parser for the Language PADL

Computation Structures Group Memo 234
February 1984

James Edward Holderle

Thesis submitted in partial fulfillment of the requirements for the S.B. degree at the Massachusetts Institute of Technology

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Parser for the Language PADL

by

James Edward Holderle

Submitted to the
Department of Electrical Engineering and Computer Science
on September 28, 1983 in partial fulfillment of the requirements
for the Degree of Bachelor of Science in Computer Science and Engineering

Abstract

PADL is a high-level digital hardware specification language that will be used as a specification tool in the FUNCHARD CAD package for the automated design of VLSI hardware for packet communication systems. Descriptions written in the PADL language are the specifications of the structure and the behavior of packet communication systems. The FUNCHARD CAD package will take as input a PADL description – the high-level specification of a system – and generate the VLSI design which implements the specified system.

The hardware generator component of the FUNCHARD package requires for its input a packet communication system specification in a data flow graph format. The purpose of the PADL parser is to begin the translation of a PADL description from the original PADL source code to an equivalent data flow graph representation. The parser performs complete syntactic checking and much of the semantic checking of the PADL source code. The output of the parser, a parse tree representation of the PADL description, is input to the PADL semantic analyzer, which completes the translation of the system specifications to a data flow graph representation.

The PADL parser is implemented in the language CLU. The parser uses the recursive-descent technique to parse all PADL constructs except for expressions; PADL expressions are parsed by the operator-precedence method. The parser support utilities – including the lexical analyzer, the output file generator, and the symbol table – are grouped into a single data abstraction. This data abstraction conveniently provides, to the parser's many recursive-descent parsing procedures, the access to the support utilities that they all require.

An important consideration in the design of the PADL parser was its ability to handle errors in its source code input. The parser detects all syntactic errors and most of the semantic errors in the source code. The parser provides, to the user, meaningful error messages that specify well the location and the nature of errors. The parser, having encountered an error in its input, is able to recover and continue its parsing in order to search for more errors in remainder of the source code. The parser has no single routine that handles all errors; errors are handled within the individual recursive-descent parsing procedures.

The PADL parser, now implemented as a independent system, can be easily modified to be incorporated as an internal component in the completed FUNCHARD package.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

Acknowledgments

The research for the design of the PADL language was supported by the National Science Foundation under research grant 7915255-MCS.

The language VAL [6], designed at M.I.T. by William B. Ackerman and Jack B. Dennis, contains many constructs (especially those for expressions and functions) that are very similar to those of the PADL language. The VAL translator, written by William B. Ackerman, was an indispensable guide in my designing and implementing the PADL parser. Several features of the PADL parser are direct imitations of those found in the VAL translator. I am greatly indebted to Bill Ackerman for the opportunity to peruse and make use of so many of the implementation techniques that I found in his VAL translator code.

I thank Willie Lim for all the time he spent with me discussing the PADL syntax, for his taking time so often to show me around the building and the computer systems, and for his putting up with my using his office.

I thank Professor Dennis for his reading this manuscript and for his offering helpful comments concerning its contents and style.

I can now, at last, thank my parents and my family for their having waited so long for me to finish.

And to my friends who have helped out so much in these past few weeks, I give many warm thanks. I could not have finished this work without their generosity.

Table of Contents

Table of Figures

Chapter One

Introduction to PADL

The language PADL (Packet Architecture Description Language) is a high-level hardware description language. PADL is intended for use as a tool in the automated design of the digital hardware for packet communication systems. Specifications written in PADL describe the structure and the behavior of packet communication systems [1, 2].

The PADL language was designed at the Laboratory for Computer Science at M.I.T. by by Clement K. C. Leung and William Y-P. Lim, members of the Computation Structures Group, under the direction of Jack B. Dennis.

1.1 The Purpose of PADL

It is planned that the PADL language be incorporated as part of the FUNCHARD system, a computer aided design (CAD) package that will aid the designer of VLSI hardware for packet communication systems [3]. Figure 1-1 shows the basic structure of the FUNCHARD system. The FUNCHARD system takes as input the high-level specification of a packet communication system and generates the design of VLSI hardware that implements the system. The designer writes, in the PADL language, the structure and behavior specifications for some packet communication system. The PADL parser and the PADL semantic analyzer translate the packet communication system specification from the PADL language to an equivalent data flow graph representation. The hardware structure generator and the silicon compiler of the FUNCHARD system then produce, from the data flow graph representation, a

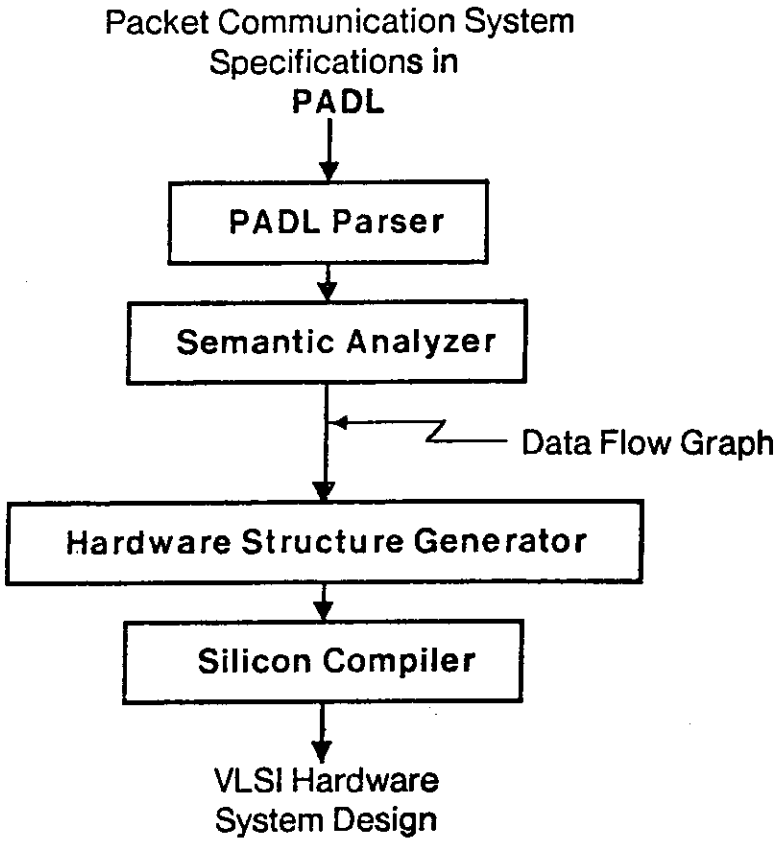

Figure 1-1:
PADL as a specification tool in the FUNCHARD CAD package.

VLSI design for the specified packet communication system.

A system specification written in PADL is a high-level description, concerned with packet communication system structure and behavior rather than with details of the structure and operation of system hardware. By using PADL to specify the structure and behavior of a system before it is implemented in hardware, it should be easier for the designer to produce systems that operate completely in accordance with their specifications. The designer, using PADL, can first specify the system on a high level, in terms of its behavior, and then generate a hardware design that correctly implements the intended system

design, instead of first designing the system hardware and deriving from the completed hardware design the actual system behavior specifications. Because the specifications written in PADL describe packet communication systems on a high level, PADL system specifications are not as strongly influenced by specific hardware technologies as are hardware-oriented system specifications. The same PADL specifications can be used to generate several different hardware implementation designs, designs using different types of hardware devices or designs with different emphases on such constraints as cost and speed. The specifications written in PADL for a system need not become out-dated or obsolete because of technological advances.

1.2 The PADL Description

The PADL language is used to write PADL descriptions. A PADL description is a textual representation of the specifications for the structure and behavior of a packet communication system. The input to the FUNCHARD CAD package of Figure 1-1 is a PADL description for the packet communication system being designed.

A packet communication system is composed of modules that operate independently and perhaps concurrently. Each module in the system has input and output ports through which it communicates with other modules in the system. Modules communicate by data packets transmitted on channels that connect the module ports. Data packets are the only means of communication between modules.

A PADL description specifies the structure and behavior of a packet communication system. The modules defined in a PADL description are independent and operate concurrently. The PADL description specifies input

and output ports for each module in the system. Each module in the PADL description is defined in terms of the behavior of the module or in terms of the module's structure.

A behavior module definition in PADL specifies the input and output ports of the module (the module must have at least one port); state variables for the module, which may be used to describe register or memory storage; what operations are performed on data packets received from which of the module's input ports; and what data packets are sent from which of the module's output ports. The operations specified in a behavior module are strictly limited to sequential algorithms. Concurrent operations within a behavior module cannot be specified.

A PADL structure module definition specifies the input and output ports of the module (each module must have at least one port), a list of modules contained within the structure module as submodules, and how the ports of the submodules and the ports of the module itself are connected. The submodules specified in the structure module definition may be any mixture of behavior or structure modules. The submodules in a PADL structure module operate concurrently.

The two kinds of module definition, behavior and structure, allow the designer of packet communication systems to specify, with different PADL descriptions, a system in varying levels of detail. A behavior module definition is a detailed specification of a module; it defines exactly a module's behavior. A structure module definition may represent a less detailed specification of the module's behavior. The PADL language supports structural composition and decomposition techniques of system design; it allows for step-wise refinement of system specifications.

A PADL description, which comprises the complete packet communication system specification, is composed of definitions written in the PADL language. There are three kinds of definitions in a PADL description: module type definitions (for both behavior and structure modules), function definitions, and data type definitions. The system designer may include, in the PADL description, definitions for functions and data types that he finds useful for specifying the behavior or the structure of the defined module types.

A PADL description is contained in one or more text files, each containing one or more PADL definitions. The files containing the PADL description are the PADL source files.

1.3 The PADL Language

1.3.1 Data Types

The PADL definition for a module type must specify the data types of the module's ports. The type of a port is determined by the type of data packets that it transmits. Each port can transmit only one type of data packet. Only ports of the same type may be connected.

PADL supports the primitive data types **integer**, **bitstr** (bit string), and **null**; and the compound types **array**, **record**, and **oneof** (tagged, discriminated union). There is, for each of the PADL data types, a complete set of operations for its use.

The hardware designed to implement a system specified in PADL must contain enough physical resources to store and manipulate the data types found in the PADL description. It is important that the amount of such resources

necessary to implement a system be discernible from its PADL description. The size of each PADL data type, primitive or complex, is static and must be explicitly specified. A PADL **array** has exactly one dimension; **array** bounds must be specified and they do not change.

PADL is a strongly typed language. The types of all PADL data elements—ports, state variables, function arguments and return values, and named data values—must be explicitly declared in the PADL description. All PADL constructs that include these data elements must contain data elements with types compatible with those required by the constructs.

Data types, primitive or compound, may be defined and referred to by name. Defined data types are useful for frequently used data types and for data types with complicated specifications. A defined data type is not a new type; it is equivalent to the PADL type that it represents.

1.3.2 Expressions and Functions

Each PADL data type represents a domain of data values. A value is a single element of data. Each value is of a particular data type.

In PADL, an expression is an abstract representation of one or more data values. The number of values that an expression denotes is defined as the *arity* of the expression. An expression of arity one denotes a single value; an expression of arity three denotes three distinct values. Each value in an expression is of a particular data type. The types of the values in an expression of arity greater than one need not be the same.

Functions in PADL, like expressions, are abstract representations of values. The invocation of a function results in an expression. Like any

expression, the expression resulting from a function invocation is of some particular, explicitly declared arity. The values of a function's resulting expression may be dependent on values provided as arguments to the function in its invocation. The values resulting from a function invocation can depend only on the values within the function definition itself and on the values passed to the function as arguments.

1.3.3 Actions

A PADL description represents specifications for a hardware system. There is, in the hardware system, some concept of the *state* of the system. The state of the hardware system is dependent on the contents of the memory storage and the contents and operational status of the module ports and their connections. PADL expressions and functions are *abstract*; they have no direct affinity with the state of the underlying system hardware. Those PADL constructs that are involved with the hardware state are the PADL actions.

Actions are a part of the operation specifications in behavior module definitions. The basic actions are the PADL constructs that receive a data packet from an input port, send a data packet from an output port, access the value of a state variable, or assign a value to a state variable. A basic action is an imperative statement, like the statements in conventional programming languages. The execution of an action directly involves the state of the hardware system; an action either modifies the hardware state, produces a value that is dependent on the state, or does both things.

The action that receives a data packet and the action that accesses a state variable both produce data values. These data values are, like those represented by expressions, of a particular, explicitly defined type. These action values can

be used in some PADL constructs in place of the values of expressions; however, any construct that contains the value from an action is itself classified as an action, for it is not abstract, but is involved with the hardware state.

1.3.4 Syntax

The syntax of PADL—which must represent module type, function, and data type definitions; expressions; actions; and all other PADL constructs—is quite extensive.

The PADL syntax is highly structured; a construct is readily identified by its position in the PADL source text or by the unique PADL keywords that delimit the more complex constructs.

The PADL syntax is very explicit; PADL is a strongly typed language, and its explicit syntax allows complete type checking within each definition in the PADL description. A definition in a PADL description may contain references to other function and module type definitions in the description. Within a definition, such a reference to an external definition must be preceded by a declaration of the external reference. The external declarations contain enough information so that each definition in the PADL description may be examined independently from the others and still be checked completely for type correctness of its constructs, including its constructs that involve external definitions.

Appendix A contains the formal definition, in a BNF format, of the complete PADL syntax.

Chapter Two

The Parser Design

The PADL parser is the first component in the FUNCHARD CAD package shown in Figure 1-1. The parser receives for its input the PADL source code, input to the CAD package, that comprises a PADL description. The function of the parser is to read through the PADL source code and to translate it from its original textual form to some representation, equivalent in meaning, that the next component of the CAD system, the PADL semantic analyzer, can more easily understand.

2.1 Functional Requirements

It is the responsibility of the parser to verify that the PADL source code that it translates is syntactically correct. The parser must perform all syntax checking of the source code so that the output that it produces for the PADL semantic analyzer represents a PADL description that is free from syntactic errors. The parser also checks the correctness of a great deal of the source code's semantics. The parser performs as much of the semantic checking as it can do without beginning to interpret the source code; for example, the parser can discover, directly from the source code syntax, the data type of an expression, but it does not attempt to evaluate an expression to ascertain its value. The semantic checking completed by the parser includes the verification of adherence to the PADL scoping rules for name identifiers and the verification of the type-correctness of all PADL expressions, function invocations, and module port operations; the parser does no checking of **array** bounds or **bitstr** lengths,

nor any other checking of limits that requires knowledge of expression values.

It is the responsibility of the parser to identify all the syntactic and semantic errors that it detects in its PADL source code input. A major consideration in designing the parser was its ability to detect, identify, and recover from errors encountered while parsing the source code. As has been already noted, it is most important that the parser detect all syntactic errors; the parser must not produce output that is not syntactically correct. It is very important that the parser, having detected an error, is able to identify the error and communicate well of its discovery to the user. The parser must generate error messages that are understandable and meaningful to the user, so that the user may easily identify, locate, and correct the errors. It is also important that the parser, having detected and identified an error in the source code, is not disabled and forced to terminate parsing. The parser should attempt to recover from every error that it encounters and should continue to check for other syntactic and semantic errors that may exist in the remainder of the code.

The PADL semantic analyzer, which continues the source code translation begun by the parser, is also apt to detect errors – semantic errors for which the parser did not check – in the PADL code. The semantic analyzer should be able, like the parser, to provide understandable error messages to the user. It is important that the parser include in its output enough information to allow the semantic analyzer to generate meaningful error messages.

2.2 Mode of Operation

The PADL parser is the first component of the FUNCHARD system to have been implemented. The current implementation of the parser is a complete, independent system, performing its own input and output operations

and interacting with the user in a friendly manner.

2.2.1 Inputs and Outputs

The inputs and outputs of the PADL parser program are shown in Figure 2-1.

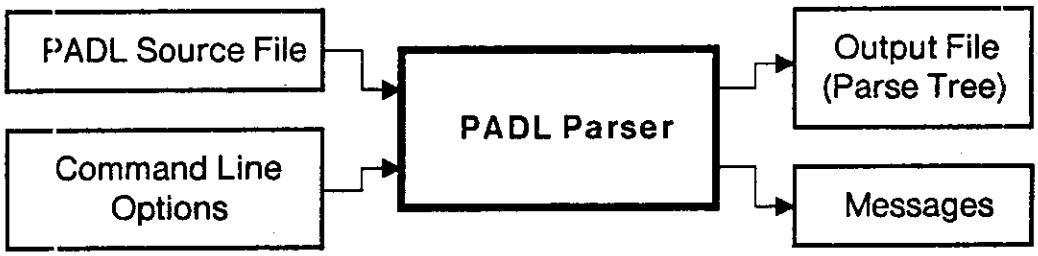

Figure 2-1:
Inputs and outputs of the parser program.

The parser receives its PADL source code input from a single PADL source file, produces a translation of the source file's contents, and generates a single output file. When invoked, the parser program requests from the user a command line. The command line specifies the name of the PADL source file to be translated and the list of options which the user selects to alter the normal behavior of the parser. The parser program's output consists of an output file and a list of the messages that the parser generates during its execution.

An entire PADL description, which represents the specification for a packet communication system, may consist of more than one PADL source file. The parser program translates only one source file. In order to translate all the definitions in a PADL description that is contained in more than one source file, the parser must be invoked once to translate each of the individual source files.

The parser produces a separate output file for each of the source files that it translates.

2.2.2 Contents of the Output File

The output files produced by the parser are the only means of communication between the parser and the subsequent segments of the FUNCHARD CAD package. The remaining segments of the FUNCHARD system have no access to the PADL source files. The parser's output files must contain all the information necessary to complete the interpretation of the PADL source code.

Each output file contains the name and creation date of the source file from which the output file was generated; the translations, in parse tree format, of all the module type definitions, function definitions, and data type definitions found in the source file; lists of name identifiers and data type specifications from all the PADL definitions in the source file; and a flag that indicates whether the parser detected any errors while translating the the source file.

A listing of the definitions for the actual data structures used in the parser's output file appears in Appendix B. Those definitions are a complete, formal specification of the contents and the structure of the output file.

2.2.3 Command Options

The basic command issued to the parser specifies only the name of the PADL source file to be translated. The suffix of the source file name must be *.pdl*. The command may include, in addition to the source file name, a list of options which modify the operation of the parser program.

In normal operation, the parser produces an output file with the same name as the input source file and the suffix ***.bnp***. An output file name, perhaps different from that of the input file, may be explicitly requested. The output file name must have the suffix, ***.bnp***. If the parser finds no file with the name (specified or assumed) for the output file, it creates a new file; if the output file already exists, its contents are replaced with the new output of the parser.

The parser normally sends the messages generated during its execution directly to the terminal display. The user may specify the name of a *list* file so that parser messages are sent to the list file, not to the terminal. If the parser finds no existing file with the specified list file name, it creates a new file; if the list file already exists, its contents are preserved and the new messages of the parser are appended to its end.

The user may choose to suppress the generation of the output file. If so, the parser translates the input source file, performs its usual checking of syntax and semantics, generates its usual error messages, but does not produce an output file. The user may choose also to suppress parser messages that list informative details of the definitions in the source file being translated.

2.3 The Function of the Parser

Many of the functional requirements that the PADL parser must fulfill (especially those concerning source code errors) were discussed in Section 2.1. This section contains a discussion of the extent of the parser's functional responsibilities. In this section are presented some of the tasks that the parser does not perform, tasks that must be performed by the PADL semantic analyzer.

The parser translates only one source file and produces an output file for

the source file translated. The source files of a PADL description that consists of more than one file must be parsed individually; there must be one parser execution for each source file translation. The parser maintains no memory of its previous executions. Each output file that the parser creates is independent from any others it has produced. The PADL semantic analyzer must have the capability to assemble the separate output files of the parsed PADL description into a unified whole; the parser makes no attempt to do so.

Each PADL source file contains one or more PADL definitions. The parser translates each PADL definition as an independent, self-contained unit. Each definition is parsed independently of all others, separately from even those definitions contained in the same source file. The parser makes no attempt to check references between different definitions in a PADL description. If a definition includes references to other definitions, it must include an external declaration for each external definition referenced. The external declarations supply all the information that the parser requires to complete its syntactic and semantic checking of the current definition. In order to verify the correctness of a construct that involves an external definition, the parser examines the corresponding external declaration that is contained within the current definition. The parser does not verify that the information in an external declaration is correct, that the information in the external declaration corresponds to the actual external definition; the parser does not attempt to verify even that a declared external definition actually exists somewhere in the PADL description. Each PADL definition is parsed separately from all others. The semantic analyzer must be able to assemble not only the individual output files, but also all the individually parsed definitions of the PADL description into a unified whole. The semantic analyzer must verify that each external declaration refers to a definition that does exist in the PADL description, and

that the information in the external declaration is correct.

The parser does not attempt to correct errors that it detects in the PADL source file. When the parser detects an error, it notifies the user of the error and then, in order to proceed with the translation of the remaining source file, it tries to recover from the error. The parser attempts to ignore the error condition by assuming that the source file contents are different; it tries to correct the erratic code. These corrections that the parser performs are in no way attempts to repair the source file; the source file is never modified. However, the parser's correction assumptions may effect the contents of the output file. When an error is detected in the source file, the parser's attempt to ignore the error condition may result in output file contents that are not accurate translations of the actual source file contents. The error flag in the output file is set to indicate that an error was detected during parsing of the input file and that the output file itself may contain errors. The semantic analyzer should not attempt to process further the contents of an output file whose error flag is set.

Chapter Three

The Implementation of the Parser

The PADL parser program is written in the language CLU [5]. The program was developed on the TOPS-20[1] operating system. The parser program should be easily transportable to any other operating system that supports CLU. All operations in the parser that interact with the operating system – input and output to the terminal and input and output to files – are performed with CLU *streams*. The names of files, whose format is dependent on the operating system, are manipulated with CLU *file_name* routines. The CLU language should deal with most of, if not with all, the differences in operating systems. Few, if any, changes to the parser program should be necessary to adapt it for execution on a different operating system.

3.1 Parsing Techniques

The PADL parser is implemented as a recursive-descent parser. The parser uses the recursive-descent technique to parse all PADL constructs except for expressions; PADL expressions are parsed using the operator-precedence method.

There are several reasons why the recursive-descent parsing technique is an appropriate choice for the PADL language. The PADL syntax is easily parsed by the recursive-descent method. The grammar that defines the PADL syntax (with the exception of that for expressions) contains no left-recursive

[1] *TOPS-20* is a trademark of Digital Equipment Corporation.

rules; a PADL construct is unambiguously identified by its first token and by its position in the source code. The parser for PADL source code requires no look-ahead and no backtracking capabilities.

A recursive-descent parser is easily implemented incrementally. The parsing procedures for syntax constructs, which comprise most of the code in a recursive-descent parser, are independent units. The parsing procedures can be added to the parser gradually as they are implemented. The recursive-descent parser being implemented, incomplete without its full complement of parsing procedures, is a fully functioning parser, although it can parse only those language constructs for which it has parsing procedures. Individual parsing procedures can be tested as they are added, one at a time, to the parser.

A recursive-descent parser can be modified readily to accept different language constructs. The independence of parsing procedures that makes easy the incremental implementation of a recursive-descent parser also makes easy the modification of the parser. The modification of one parsing procedure should not make necessary the modification of others. The syntax specifications of PADL were changed as the PADL parser was being implemented; they are apt to be further modified as the remainder of the FUNCHARD system is developed and the PADL language is actually used. It is important that the PADL parser can be easily modified to accept changes and additions to the PADL syntax.

A recursive-descent parser has the *valid prefix property*: the parser can detect an error in the source code as soon as it encounters code that is not the beginning of a valid construct. Not all parsers have the valid prefix property; an operator-precedence parser, for example, may read far past a mistake in the source code before it detects the error. The valid prefix property of a recursive-

descent parser improves the parser's ability to handle errors in the source code. The exact location of an error is known and can be told to the user. An error can be identified and repaired before any more code is parsed; the parser never needs to unparse and parse again the code, following a mistake, that was translated before the error was detected.

The grammar for PADL expressions is full of left-recursive rules. Attempting to parse PADL expressions using the recursive-descent technique would be very difficult. The grammar for PADL expressions is, however, a true *operator grammar*: any two terms in a PADL expression must be separated by an operator; no two terms may appear together. The syntax for PADL expressions is easily parsed with the operator-precedence method. The PADL parser therefore uses the operator-precedence technique to parse the parts of PADL expressions concerned with operators; the remaining PADL constructs, including the terms in expressions, are parsed using the recursive-descent technique.

The PADL parser is a one-pass parser; it scans the input file only once in order to parse the source code. In all situations (with one exception), when the parser encounters a name identifier in the PADL source code, it knows already, from having examined the preceding code, the translation value of the name. All name identifiers used in the PADL code must be explicitly declared. Only one PADL construct allows the use of a name identifier in the code to precede the name's declaration. A discussion of the construct, and the manner in which it is parsed in one pass, appears in Section 3.5.3.

3.2 The Parser Data Abstraction

The PADL parser consists of a set of parsing procedures and a set of parsing support utilities. The support utilities provide the parser's necessary functional capabilities, such as reading the source file, creating the output file, and generating error messages. The parsing procedures use the support utilities in order to perform their parsing functions.

Each of the parser's parsing procedures must have access to the parser's support utilities. In the CLU language, a procedure has access to only those items passed to it as arguments. Each of the parsing procedures, a CLU procedure, must receive as arguments all the parser's support utilities. The number of parsing procedures in the PADL parser is large; the number of parsing procedure calls in the parser is larger still. In order to avoid the task of listing explicitly several times, as arguments in each of the many parsing procedure calls, the entire list of parser support utilities, all of the parser's support utilities are grouped together into a single data abstraction.

The **parser** data abstraction contains the parser's support utilities. The functions that the **parser** abstraction supports are shown in Figure 3-1. The **parser** abstraction is passed as the first argument to each of the parsing procedures. It provides a convenient means through which to give all the parsing procedures access to the support utilities. Grouping all the parser's support utilities into one data abstraction makes explicit the separation and the interface of the utilities and the parsing procedures. Because the **parser** is a data abstraction, the only means of access to the contained support utilities is through the **parser** abstraction's defined routines. The **parser** abstraction decreases the interdependence of the utilities and the parsing procedures. If the **parser** access routines remain unchanged, the support utilities within the **parser** abstraction

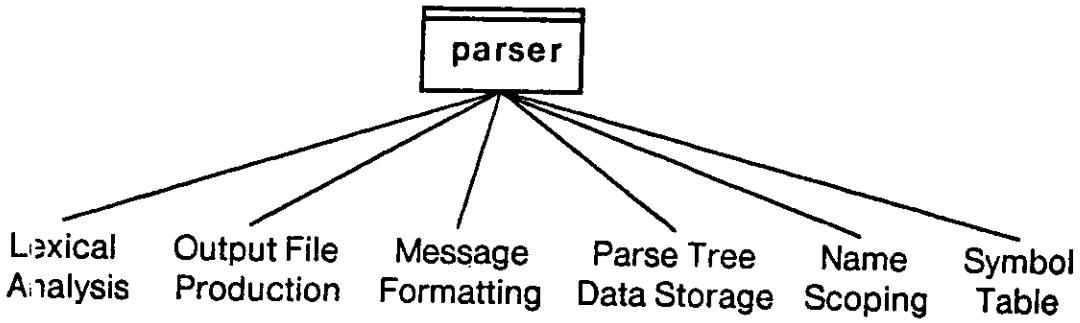

Figure 3-1:
Functions of the **parser** data abstraction

can be modified without making necessary the modification of all the parsing procedures that use them.

A discussion of the support utilities contained in the **parser** abstraction follows.

3.2.1 The Lexical Analyzer

The **tokrdr** (token reader) data abstraction is the lexical analyzer for the PADL parser. It is contained in and accessed through the **parser** data abstraction. The **tokrdr** reads the PADL source code from the parser's input file. It is the parser's only means of access to the source file.

The **tokrdr** reads characters from the source file; ignores insignificant blank spaces, tabs, new line characters, and comments; and recognizes groups of characters that constitute valid PADL tokens. PADL tokens are punctuation and operator symbols (having one or two characters), **integer** constants, **bitstr** constants (in binary, octal, or hexadecimal format), PADL keywords, and name identifier strings. When the **tokrdr** recognizes a token in the source file, it creates

a *token* object. A *token* object identifies the type and the value of a token, and, if the token is a name identifier string, denotes the current name bindings of the identifier.

The **tokrdr** lexical analyzer contains two subordinate data abstractions: the **chrdr** and the **line_buff**. The **line_buff** (line buffer) abstraction reads the source file; it contains the *stream* through which the source file is read. The **line_buff** keeps the characters of one source file line in a buffer with a pointer to the character that is next to be read. When all of the characters in the buffer have been read, the **line_buff** reads into the buffer the next line from the source file. The **line_buff** counts and assigns numbers to the source file lines. The **chrdr** (character reader) abstraction contains the **line_buff**. The **chrdr** makes simple the interface between the **tokrdr** and the **line_buff**; it supplies to the **tokrdr** the source file characters, one at a time.

The **tokrdr** provides two basic routines, *peek* and *flush*, which give the parsing routines access to the source file tokens. The *peek* routine does not read from the source file; it returns the *token* object for the token last read. Subsequent calls to *peek*, before an intervening call to *flush*, will return the same *token* object. The *flush* routine reads characters from the **chrdr** to identify the next token in the source file. The *peek* calls following a *flush* will return the *token* object for the new token; the previous *token* object is discarded and can be accessed no more.

The PADL lexical analyzer needs no look-ahead capabilities to identify tokens. The last character in a PADL token is readily identified when the character itself or the character following it is read. Neither the **chrdr** nor the **line_buff** supports look-ahead. Like the **tokrdr**, the **chrdr** abstraction provides *peek* and *flush* routines with which the **tokrdr** can access the source file

characters. The **chrdr** *peek* routine allows the **tokrdr** to identify the end of a token by viewing, and not discarding, the character following the token.

The **tokrdr** abstraction contains and maintains the symbol table for the PADL parser. The symbol table is discussed in Section 3.3.

The PADL lexical analyzer makes no attempt to detect errors in the spelling of identifier names or PADL keywords. Each identifier encountered in the source file is assumed to be a valid identifier name and is listed in the symbol table. Because the PADL syntax requires that the binding of every identifier name be explicitly declared, any errors in the spelling of identifier names should be detected in the parsing procedures, which can identify all invalid uses of an undeclared (possibly misspelled) identifier name.

3.2.2 Output File Production

The **bnper** data abstraction, part of the **parser** abstraction, controls the creation of the parser output file. The parser output file is a CLU image file. It is written and must be accessed through a CLU *istream*. The use of an *istream* to access the output file is no more difficult than would be the use of a normal CLU *stream*. A parser output file written as an image file is much smaller than the same output file written as a normal CLU data file; a CLU image file contains no explicit CLU data type specifications intermixed with the actual data. The use of the image file instead of the normal data file for the parser output file introduces no real dangers of data misinterpretation. The output file is written by the parser is and read by the semantic analyzer; the user has no need to access – to interfere with – the output file contents. The contents of the parser output file are highly structured; it is not likely that the semantic analyzer's attempt to read a corrupted output file could be successful.

The **bnper** abstraction (named for the parser output file ***.bnp*** suffix) performs two functions. The **bnper** generates the output file. It contains the *istream* to the output file. It sends the output data, in the correct output file format, through the *istream* to the output file. Because the production of the parser output file is controlled within a data abstraction, only the **bnper**, and not the rest of the parser, need be changed if the structure of the output file must be altered.

The structure of the output file requires that all the file contents be sent through the output *istream* at one time, when the parsing of the entire PADL source file is complete. The individual definitions in a PADL source file are parsed separately. The **parser** abstraction is used to translate only one PADL definition at a time. The **bnper** provides storage for the translations of PADL definitions. When the parsing of a definition is complete, the definition's translation is stored in the **bnper** abstraction until all the other definitions in the source file are parsed. When the entire source file is parsed, the **bnper** generates the output file.

3.2.3 Formatted Messages

The messages generated by the parser are sent to the user through the *list stream*. The *list stream* connects the parser with either the user's terminal or with a message output file. The *list stream* is contained in the **msger**, another part of the **parser** abstraction. The **msger** (messenger) data abstraction controls the format of the parser messages. The **msger** formats and sends through the *list stream* all the different kinds of parser messages: error messages, warning messages, and messages providing helpful helpful diagnostic information. Because the formats of the parser messages are controlled within the **msger** abstraction, only the **msger** need be changed if the format of any parser messages

must be modified.

The most important function of the **msger** is its generation of error messages. An important functional requirement of the PADL parser is that its error messages be meaningful and easy to understand. An error message should make clear the nature of the error and should indicate the error's exact location in the source file. A sample of the parser's error messages, showing the format of two messages generated by the **msger**, is shown in Figure 3-2.

```
ERROR AT LINE 35:
 type packet = record[ address: bitstr[0:7], value: pval ]
                                              -↑-
A ';' must separate record field specifications.  Assuming
this is a ';'.
     * Resuming translation...


ERROR AT LINE 206:
 let x: integer = from inport  im  x + sum,
                              -↑-
Expecting "in" to begin LET construct body.
     * Skipping to --

LINE 208:
   endlet
  -↑-
     * Resuming translation...
```

Figure 3-2:
Sample error messages

Each error message generated by the **msger** indicates the source file line number on which the error occurs. It includes the text of the line containing the error and a pointer to the exact position of the error. The message explains clearly what type of error is being indicated and how the parser intends to handle the error.

The information that specifies the the error's location and context is obtained from the **line_buff** abstraction (Section 3.2.1). The **line_buff** returns (through the **tokrdr**) the *place* objects that are used by the **msger** to depict the location of errors. A *place* object contains a string containing the characters in a source file line, a pointer to a character in the line, and the line number of the source file line represented.

3.2.4 Parse Tree Data Storage

The **parser** abstraction contains data structures to hold the parse tree representation being constructed for the PADL definition being parsed. The parser must be able to translate the three kinds of PADL definition: module type definitions, function definitions, and data type definitions. Each kind has a different parse tree representation. The **parser** abstraction contains data structures to hold the parse tree representations for all three kinds of PADL definition, along with an indicator that denotes what kind of definition is being translated.

The parse tree storage structures contain data for only one PADL definition, the definition being currently parsed. The completed translations of the definitions in the file that were previously parsed are stored in the **bnper** abstraction (Section 3.2.2).

3.2.5 Name Scoping Procedures

The PADL language specifies rules that define the scopes of bindings to name identifiers. There are several different kinds of name bindings in PADL; each kind has defined for it a different scoping rule. The **parser** abstraction provides routines and data structures that the parsing procedures use to implement some of the name scoping rules. The scoping rules that are easily

implemented are accomplished entirely within the parsing procedures.

A more detailed discussion of the PADL scoping rules for name identifiers and how the parser implements them is contained in Section 3.5.

3.3 The Symbol Table

The symbol table of the PADL parser is used to store information regarding the name bindings of the name identifiers in a PADL definition. In PADL, a name identifier may be bound to a data value (an abstract value or a state variable), a function, a data type specification, a **record** field, or a **oneof** tag. A single name identifier may have (at the same time, without conflict) more than one or all of these bindings. The information for the bindings of **record** field names and **oneof** tag names is not stored in the symbol table. The manner in which **record** field and **oneof** tag name references are resolved is discussed in Section 3.4.

The symbol table is contained in the **tokrdr**, a part of the **parser** data abstraction (Section 3.2.1). The symbol table is a CLU *table*; it is a hash table data structure. There are entries in the symbol table for name identifiers and for the PADL reserved words. PADL keywords are reserved words; they cannot be used as name identifiers. The symbol table entry denoted by a PADL reserved word string indicates that the string is a reserved word. The symbol table entry for a name identifier contains the *symbol* object for the identifier. A *symbol* object is a data structure that records a name identifier's current bindings to data values, functions, and data types. Each name identifier found in the PADL definition has exactly one entry in the symbol table; there is a unique *symbol* object for each identifier name.

The keys for the symbol table entries are identifier strings. The case (upper or lower) of the characters in PADL reserved words is not distinguished; the symbol table keys for the PADL reserved words are lower case strings. The case of the characters in name identifiers is distinguished. The symbol table keys for name identifiers are strings containing both upper and lower case characters.

The **tokrdr** accesses and updates the symbol table. When the **tokrdr** encounters an identifier string in the source file, it accesses the symbol table to ascertain the identifier's name bindings. Using as a key the identifier string, converted to all lower case characters, the **tokrdr** checks first for a symbol table entry denoting a reserved word. If the identifier is a reserved word, the **tokrdr** simply creates a *token* object (Section 3.2.1) that signifies the particular PADL keyword. If the indicated symbol table entry is not for a reserved word, or if no entry having the lower case string key exists, the **tokrdr** again checks the symbol table, using as a key the original identifier string with its correct capitalization. If the identifier string does denote a symbol table entry, the **tokrdr** creates a *token* object that contains the name identifier's *symbol* object. If there is no entry in the symbol table for the identifier, then the identifier has not yet been encountered while parsing the current definition. The **tokrdr** creates a new *symbol* object for the name identifier. The new *symbol* object indicates that the name identifier has no current bindings. The **tokrdr** creates, with the new *symbol*, a new symbol table entry and a *token* object for the new name identifier.

The **tokrdr** never modifies the *symbol* objects stored in the symbol table. The **tokrdr** gives the parsing procedures access to *symbol* objects through the *token* objects for name identifiers. The parsing procedures can, through the *token* object, modify the *symbol* object for a name identifier in order to change its binding information.

The PADL parser parses separately the individual definitions in a source file; the parser requires a separate symbol table for each definition that it parses. The parser's symbol table must contain entries for only those name identifiers from the definition being currently parsed. Name identifiers appearing only in previously parsed definitions should not have entries in the current symbol table. The **tokrdr** provides a routine, *refresh*, that must be called before the parsing of each definition in the source file. The *refresh* routine creates a new symbol table for the **tokrdr**. The new symbol table contains entries for the PADL reserved words only. The **tokrdr** adds symbol table entries for name identifiers as the identifiers are encountered in the definition being parsed.

The parser translates all name references. The parser output file, containing the parse tree representations for PADL definitions, has in it no unresolved name identifier references. Name identifier references in the source code are encoded in the parser output file as integer indices to arrays, arrays that contain representations of objects appropriate to the name identifier references, such as the translations of function, module type, or data type definitions. Because the parser implements the PADL rules for name identifier scoping and resolves the references to all name identifiers in the source file, the PADL semantic analyzer has no need of the parser's symbol table. The hash table symbol table is not included in the parser output file.

In addition to the hash table symbol table, the **tokrdr** contains a list of identifier name strings. This *name list* contains all the identifier strings from all the definitions in the source file. The *name list* contains both the identifier strings of those name identifiers with entries in the symbol tables and the identifier strings of those names used as **record** field names and **oneof** tag names. The strings in the *name list* are not in any sorted order, and the same name string may appear in the list more than once.

The *name list* array is included in the parser output file. It is the only data structure in the output file that contains identifier name strings. In the output file, all name identifier strings are represented by integer indices to this array. The external references (between separate definitions), which the PADL parser does not attempt to resolve, must be resolved by the PADL semantic analyzer. The semantic analyzer needs name identifier strings in order to match external declarations with the actual PADL definitions to which they refer. The inclusion of the *name list* in the output file also allows the semantic analyzer to provide, to the user, meaningful messages concerning name identifier references. The indication of the actual identifier in the encoded translation of a resolved name reference is unnecessary; the PADL semantic analyzer does not need to know the identifier string for a name reference that the parser already resolved. The parser, however, encodes into the translation of every resolved name reference the *name list* index of the identifier string used in the reference. The semantic analyzer can use the actual name identifier strings to generate meaningful error messages concerning name references.

3.4 The Type Specification List

The parser stores all information concerning the type specifications for PADL data elements in a single data structure, the type specification list. PADL is a strongly typed language. In the PADL source code, the type specifications for the data elements—module ports, state variables, function arguments and return values, and named data values—must be explicitly declared before the data element may be used in PADL constructs. The parser can discern the type specification for the resulting data values of all PADL operations and expressions by examining the type specifications of the data elements used in composing such constructs.

The parser needs access to the information in the type specification list in order to perform its type checking of the PADL source code. The parser must verify that the type specifications of data elements contained in PADL constructs conform with the type specifications required by the constructs. The subsequent segments of the FUNCHARD system also need the information in the type specification list. The type specification of a data element indicates the amount of physical resources required to store and to manipulate the data element. The semantic analyzer needs access to the information in the type specification list in order to complete the type checking of the PADL source code; the semantic analyzer, not the parser, must verify the type correctness of external references between the separate definitions in the PADL description.

The type specification list is a part of the **parser** data abstraction in the PADL parser (Section 3.2). All of the parser's parsing procedures have access to the type specification list. The parsing procedures read the type specification list in order to check the type-correctness of the constructs that they parse; they modify the type specification list by adding to the list new type specification entries for the new data elements whose declarations they encounter. The type specification list is included in in the parser output file so that the remaining components of the FUNCHARD system may have access to the information in the list.

3.4.1 The Structure of the List

The type specification list is an array of *typespec* objects. Each *typespec* object represents the data type specification of a PADL data element. In the parser's parse tree translations of PADL definitions, all data type specifications are represented by indices to the type specification list. The data type of each data element is represented by an integer, the index to the type specification list

for the *typespec* object representing the element's data type.

The first entries in the type specification list are reserved for the *typespec* objects that represent the PADL primitive types **null**, **integer**, and **bitstr[1:1]**. These primitive types appear frequently in the PADL source code. The parser's parsing procedures know the indices of these primitive types; the same indices are always used to represent them.

The parser produces and maintains only one type specification list. The translations of all the definitions in a PADL source file contain index references to the same type specification list. New *typespec* entries are added to the type specification list for every new data element (with the exception of those data elements having one of the primitive types mentioned above) encountered in the source code. The translations of different PADL definitions from the same file do not contain references to the same type specification list entries. The *typespec* objects for a single PADL definition are located in a contiguous segment of the type specification list, apart from those of any other definition.

A *typespec* object contains the complete type specification for a data type. The PADL data types are **null**, **integer**, **bitstr**, **array**, **record**, and **oneof**. A *typespec* object also indicates the data type's name, if it exists. (Only user-defined data types, defined in a **type** construction, have names.)

A *typespec* object includes all the type specification information that is relevant for the particular data type that it represents. For the PADL primitive types **null** and **integer**, the data type indication alone suffices for a complete type specification. The types **bitstr** and **array** require the indication of two integer-valued expressions for their upper and lower bounds. The PADL compound types require type specifications for the subtypes from which they are composed.

The *typespec* object for an **array** type must indicate the data type of the **array**'s elements; the *typespec* objects for **record** and **oneof** types must indicate the names and the data types of the **record** fields and the **oneof** tags. The type specifications for the subtypes of compound data types are, like all data types, represented by indices to the type specification list. The complete type specification of a compound type, such as a **record**, may involve several of the *typespec* objects in the type specification list. The type specification of a **record** requires a *typespec* object for the **record** type itself; the top-level *typespec* contains the indices of the *typespec* objects for the data types of the **record** fields; and, if the data types of the **record** fields are themselves compound types, their *typespec* objects contain the indices of even more *typespec* objects.

The *typespec* representation for the fields in a **record** type is contained in two arrays. The first array is a list of indices to the parser's *name list* (Section 3.3) that denote the name strings for the **record** fields. The indices are listed in *canonical* order, corresponding to the alphabetical ordering of the field name strings. The second array contains a list of indices to the type specification list. These indices indicate, in the same canonical order, the data types of the **record** fields. The type specifications for the tags in a **oneof** type are represented in a similar manner. The *typespec* object for a **oneof** type contains, to represent the **oneof** tags, an array of *name list* and an array of type specification list indices. The indices in both arrays are listed in canonical order, corresponding to the alphabetical ordering of the tag name strings. A **oneof** *typespec* object may also include an array of integers or an array of strings to indicate the optional **integer** or **bitstr** encodings for the **oneof** tag names; this array, if it exists, is also arranged by the canonical ordering of the tag names.

The canonical ordering of **record** fields assigns to each field in a **record** a unique integer, the index of the field in the *typespec* canonical field arrays, by

which it can be identified. In the parser's translation of PADL definitions, a reference to a **record** field is represented by two integer indices. The first is an index to the type specification list; it denotes the *typespec* object for the specification of the **record** type. The second is an index to the two canonical field arrays in the specified *typespec* object; it denotes the chosen **record** field. The parser represents **oneof** references in this same manner. A **oneof** tag is represented by the type specification list index of the **oneof** *typespec* object and by the index to the *typespec* canonical tag arrays that denotes the chosen **oneof** tag.

In the PADL source code, the use of name identifiers in reference to **record** fields or **oneof** tags is limited to the few constructs that involve **record** or **oneof** type specifications. It is quite clear, in the context of these type specification constructs, which name identifiers are used in reference to **record** field names and **oneof** tag names. The binding information of **record** field and **oneof** tag names is not stored in the parser's symbol table (Section 3.3). The canonically ordered field and tag arrays in the type specification list *typespec* objects are used to resolve references to **record** field and **oneof** tag names. When, within a type specification construct, it encounters a name in reference to a field or a tag, the parser looks through the canonically ordered field or tag name array in the appropriate *typespec* object. If the encountered name identifier matches an entry in the canonical array of *name list* indices, the parser knows that the identifier is bound to the **record** field or to the **oneof** tag denoted by the canonical array index. If the encountered name identifier is not found in the canonical *name list* index array, the parser knows that the identifier is not bound to a **record** field or **oneof** tag name. The resolution of identifier bindings to **record** field and **oneof** tag names is easily accomplished using the canonical *name list* arrays in the *typespec* objects. There is no need to clutter the parser's

symbol table with the information for these **record** and **oneof** field and tag name bindings.

Adherence to the PADL scoping rules for name identifier bindings to **record** field and **oneof** tag names occurs automatically. An identifier name used as a **record** field or **oneof** tag has the same scope as the type specification of the **record** or **oneof** to which it belongs. All *typespec* objects that are referenced represent type specifications that are defined within the current scoping block of the PADL source code. If a *typespec* object is referenced, the type specification that it represents is defined and the name identifiers for its **record** fields or **oneof** tags are bound in the current scoping block. A valid reference to an unbound field or tag name identifier cannot occur.

3.5 Scoping of Name Identifiers

A scoping block in the PADL source code is a PADL construct that can contain the declarations of name identifiers. Each name identifier in the source code is associated with the scoping block in which it is declared. The declaration of a name identifier results in a new binding for the identifier. The binding of a name identifier exists only within the scoping block of its declaration.

The *symbol* objects in the parser's symbol table (Section 3.3) denote the current bindings of name identifiers to data values, function definitions, and data type definitions. The information in the symbol table must be valid always; a *symbol* object must always indicate the current bindings of a name identifier. When, while scanning the source code, the parser encounters a new scoping block, it must modify the symbol table to reflect all the new name bindings resulting from the name identifier declarations contained within the block. When the parser encounters the end of a scoping block, it must modify the

symbol table to undo the bindings of any name identifiers that were declared within that block. Upon exiting a scoping block, the bindings of the name identifiers declared within the block no longer exist; any information denoting the non-existent bindings must be removed from the symbol table.

The parser assigns a unique number to each scoping block that it finds in a PADL definition. The block number is used to identify the scoping block itself and all the name identifier bindings that are declared within the block. For example, the parser numbers all the data value names defined in a scoping block; the first value name declared is numbered one, the second is numbered two, and so on. In the parser's translation of PADL definitions, a data value name binding is represented by two integers: the number of the scoping block in which the value name declaration appears and the number of the value name within that scoping block.

In the PADL source code, scoping blocks may occur nested within other scoping blocks. The PADL scoping rules, which define whether a name identifier binding exists within nested scoping blocks, differ for name identifier bindings to data type definitions, function definitions, and data values. The different PADL scoping rules and the manner in which the parser implements the different rules are discussed below.

The **parser** data abstraction (Section 3.2) maintains data structures and provides routines that help the parsing procedures to implement the PADL name identifier scoping rules. The **parser** abstraction has a counter that generates a unique number for each scoping block and a counter that indicates the current depth of scoping block nesting. The **parser** abstraction maintains a stack of scoping frames and provides a set of routines to make use of them. Each scoping frame contains the scoping information for a single PADL scoping

block. A scoping frame contains, in addition to the scoping block and nesting level numbers, information concerning the name bindings of data values, function definitions, and data type definitions that are declared within the scoping block. When a parsing procedure encounters, in the source code, the beginning of a scoping block, it calls the **parser** abstraction's *enterscope* routine. The *enterscope* routine creates a new scoping frame—one that exhibits no name bindings—and pushes the new frame onto the scoping frame stack. The **parser** abstraction provides routines that record name identifier bindings in the current (on top of the stack) scoping frame; the parsing procedures call on these routines as they encounter name identifier declarations in the current scoping block. When a parsing procedure encounters the end of a scoping block, it calls the *exitscope* routine, which pops the current scoping frame from the stack and unbinds all the name bindings recorded in the frame. The following sections explain in more detail the **parser** abstraction's scoping frames, the associated scoping routines, and how the parsing procedures use them to implement the PADL scoping rules.

3.5.1 Data Type Names

The scope of a name identifier binding to a data type definition is the entire scoping block in which the binding is declared, including all scoping blocks nested within that block. A name bound to a data type definition may not be bound again to a data type definition within the scoping block of its declaration. The redefinition, in a nested scoping block, of a data type name identifier already declared in an outer scoping block is not permitted.

The scoping rule for data type definition bindings is easily implemented. When a parsing procedure encounters a data type definition, it calls the **parser** abstraction's *add_lcltyp* (add local data type definition) routine. The *add_lcltyp*

routine first checks the current binding of the name identifier being bound to a data type specification. It is an error if the name identifier is already bound to a data type definition. If the name is unbound, *add_lcltyp* modifies the *symbol* object for the name identifier to reflect its new data type binding and stores the modified *symbol* in the current scoping frame. When a scoping block is exited and its scoping frame is popped from the stack, the bindings of the data types defined in the scoping block must be undone. The *exitscope* routine modifies all the *symbol* objects for the locally defined data types, which were saved in the scoping frame, so that they denote no bindings to data type definitions.

3.5.2 Function Names

An *external* function is a function whose definition appears by itself, not within another PADL definition. An external function definition is one of the three kinds of definition of which a PADL description is composed. An external function may be invoked in any of the other definitions in the PADL description; a definition that makes use of an external function must include an external function declaration.

The scope of a name binding to an external function is the entire PADL definition in which it is declared as an external function, including all nested scoping blocks. When a parsing procedure encounters an external function declaration, it modifies the *symbol* object for the function name identifier to record the binding to the external function. The binding to an external function is permanent—it endures throughout the entire PADL definition; the function binding information in the modified *symbol* object is not changed during the subsequent parsing of the PADL definition.

An *internal* function is one whose definition appears within a PADL

definition – within another function or within a module type definition. The scope of a name binding to an internal function definition is the scoping block in which the function is defined. A name binding to an internal function is not valid within the function definition itself nor within the other internal functions defined within the function definition. The scope of a name identifier binding to an internal function definition is the scoping block immediately enclosing the internal function definition.

The **parser** abstraction provides, to the parsing procedures, the same support for implementing the scoping of internal function name identifiers as it provides for the scoping of data type definitions. The **parser** abstraction's *add_lclfun* (add local internal function definition) routine modifies the internal function binding information of the *symbol* objects for internal function name identifiers and saves the modified *symbol* objects in the current scoping frame. A parsing procedure, when it encounters an internal function definition, saves the *symbol* object for the internal function name. Without yet calling the *add_lclfun* routine, the parsing procedure continues and completes the translation of the internal function definition. When the parsing of the internal function definition is completed, the procedure calls *add_lclfun* to bind the function name identifier to the internal function definition just parsed. By waiting until the function definition itself is parsed before binding its name identifier, the PADL scoping rule for internal functions is correctly implemented. The function name identifier is bound within the scoping block immediately enclosing the function definition, but not within the function definition itself. When a scoping block is exited and its scoping frame is popped from the stack, the bindings of the internal functions (like those of the data type definitions) defined in the scoping block must be undone. The *exitscope* routine modifies all the *symbol* objects for the locally defined internal functions, which were saved in the scoping frame, so

that they denote no bindings to internal definitions.

A name identifier can be bound to at most one function definition (external or internal) at any one time. The attempt to bind an identifier name, already bound to a function, to another function definition is an error.

3.5.3 Data Value Names

There are three types of data value names in PADL to which identifiers may be bound: state variables, formal parameters of functions, and abstract value names (such as those in the **let** construct). The scope of a binding to a data value name identifier is the entire scoping block in which the binding is declared, except for internally nested scoping blocks in which the same name identifier is declared as another data value. Unlike the name identifiers bound to data type and function definitions, a name identifier bound to a data value may be redeclared and bound again inside internally nested scoping blocks. The binding of a referenced value name identifier is the binding resulting from its declaration in that scoping block, which, of all the blocks containing declarations of the value name, most closely surrounds the value name reference.

In the PADL source code, the declaration of a data value name indicates the type specification for the data type of the value name; the definition specifies a value name's actual data value. It is the declaration that results in the binding of an identifier to the data value name. A value name must be declared before it is defined. A value name must be defined before it is referenced for its value. In a *symbol* object, the information concerning binding to a data value name indicates whether the bound value name is declared only or if it is defined as well as declared. The parser verifies that no value name is referenced for its value before the data value name is both declared and defined.

The implementation of the PADL scoping rule for data value names is more difficult than that for data type and function definitions. Value name bindings may be redefined within nested scoping blocks. If a name identifier is redefined inside an inner scoping block, the identifier has a new, different data value binding within the inner scoping block. Outside the inner scoping block—before it and after it—the identifier has its other, older binding. The scanning parser, when it enters an inner scoping block, must modify the symbol table information for the name identifiers declared therein so that their *symbol* objects reflect their new (inner scope) data value bindings. When the parser exits an inner scoping block, it cannot simply unbind the value name bindings declared within the block; the parser must restore, in the symbol table, the information for the declared identifiers' previous (outer scope) value name bindings, the value name bindings that existed before the inner scoping block was entered.

In the scoping frames of the **parser** data abstraction, the information for an identifier's data value name binding indicates not only its present, but also its previous value name binding. When a parsing procedure encounters a data value name declaration, it calls the **parser** abstraction's *add_lclval* (add local data value) routine. The *add_lclval* routine modifies the *symbol* object for the name identifier so that the *symbol* object indicates the newly declared value name binding. The identifier's previous value name binding is stored in the current stack frame along with its modified *symbol* object. The *exitscope* routine, called when the parsing procedure reaches the end of a scoping block, modifies all the *symbol* objects for the identifiers declared as value names within the scoping block; it replaces the value name binding information in the *symbol* objects with the previous value name binding information, which was stored in the scoping frame. The *exitscope* routine restores to the *symbol* objects the binding

information for the value names to which the identifiers were bound before the inner scoping block was entered.

The parser must handle the formal parameters of a function differently than the other kinds of data value names—the state variables and the abstract value names. A function's formal parameters are declared in the header of the function definition. The name identifiers declared as formal parameters are defined (by the actual parameters) when the function is invoked. The definition of the formal parameter value names is not explicit in the code of the function definition; the parameters are declared but not defined in the function header. The parser, in order to translate a function definition as if it had been invoked, handles the parameter declarations in a function header as if the parameter value names were defined as well as declared. The parser, when modifying the *symbol* object for a name identifier declared as a function's formal parameter, records binding information that denotes a defined data value name.

The translation of formal parameter name declarations presents a special problem to the parser. In PADL, the header of a function (including the formal parameter declarations) is the first component in the function definition. The definitions of data types, which can be used as type specifications throughout the function definition, are listed after the function header. The PADL language allows the data types defined in a function to be used in the declarations of the function's formal parameters. The parameter declarations, which may reference the defined data type specifications, precede the data type definitions. The PADL parser, translating the source code in one pass, must be able to translate the declaration of a formal parameter that makes use of a data type specification whose definition has not yet been encountered in the source code. The name identifier for such a data type definition is not yet bound to the type specification when it is referenced in the formal parameter declaration. The

formal parameter declaration is the only PADL construct in which a name identifier may be referenced before it has been explicitly declared.

The parser must perform backpatching in order to translate, in one pass, the formal parameter declarations of functions. The **parser** abstraction contains an array of *symbol* objects, the *undefined* list. If, while translating the heading of a function, the parser encounters a name identifier used as a type specification and the identifier has, at that time, no binding to a data type definition, the parser assumes that the identifier is the name of a data type that has not yet been defined. The parser adds, to the type specification list (Section 3.4), a new *typespec* entry that represents the undefined type specification. The parser modifies the *symbol* object for the data type name identifier so that it denotes a data type binding to the new *typespec* object. The parser adds the modified data type *symbol* object to the *undefined* list in the **parser** abstraction. The definition of the parameter name can then be recorded—in the parameter name's *symbol* object and in the current scoping frame—using the modified data type name *symbol* object, which denotes a binding, though undefined, to a data type definition. When the parser translates a data type definition in the function body, it checks the *undefined* list to see if the name identifier being bound to a data type definition was previously encountered in the function's formal parameter declarations. If the name identifier is included in the *undefined* list, if it was referenced as a data type in the function header, the parser modifies the existing type specification list *typespec* entry for the data type. The *typespec* object, which represents an undefined data type specification, is changed so that it represents the actual data type specification indicated in the definition of the data type.

In brief, if the correct type specification binding for a data type name is not yet known when the parser references the name in a formal parameter

declaration, the parser creates a slot—an undefined type specification—to use temporarily as the data type name binding in the translation of the parameter declaration. The slot represents a type specification that is not yet defined. When the actual definition of the data type name is later translated, the slot, which was used in lieu of the actual data type name binding, is changed to the real data type binding indicated in the actual data type definition. This backpatching technique allows the parser to translate, in one pass, the declarations of formal parameters.

3.6 Error Recovery

An important functional requirement of the PADL parser is that it, having encountered an error in its source code input, be able to recover from the error and continue the parsing of the entire source file. The PADL parser has no centralized error handling utility that accomplishes the recovery from errors; it has no single routine that recovers from all possible errors. Instead, in the parser, errors are handled completely within the individual parsing procedures. Reporting and recovering from an error is the responsibility of the parsing procedure that detects the error.

By handling errors within the individual parsing procedures, the parser is able to handle errors better than if all errors were handled in a single routine. Because a parsing procedure is so familiar with the correct syntax of the construct in which it detects an error, it is more likely to identify and recover from an error than a central error handling routine. Each parsing procedure, which parses a single PADL construct, knows the exact syntax of the construct that it parses. A parsing procedure can be aware of errors that are likely to be found in its construct, and can handle those errors especially well. The parsing procedure can check for errors likely to occur in its construct; it can readily and

accurately identify the common, expected errors. Even the less expected errors can be handled better in the parsing procedures than in a single error routine. A parsing procedure is more likely to recover from an error; it is familiar with the syntax of its construct and is more likely to be successful in modifying the incorrect source code to arrive at a syntactically valid construct. A single, centralized error handling routine would not know the complete syntax of all constructs in which errors occur.

A single error handling routine would offer, at best, only generalized methods of error identification and recovery. Because the PADL parser handles detected errors directly within the parsing procedures, each different possible error can be recognized and handled in the best possible way. The parsing procedures can identify and recover from each error, not in a general way, but in the manner best suited to the handling of each particular error.

The **parser** data abstraction provides, to the parsing procedures, routines that send error messages to the user (Section 3.2.3). When a parsing routine detects an error in the source code, it sends an error message that indicates the location and the nature of the error. An error message also indicates the manner in which the parsing procedure is attempting to recover from the error. Figure 3-2 on page 31 shows the format of the parser's error messages.

Although each parsing procedure handles errors in its own, individual manner, all parsing procedure methods comply with the following general method of error recovery. To recover from an error, a parsing procedure first attempts to remove unnecessary text, add missing text, or modify existing text in the source code in order to create a valid PADL construct. If the parsing procedure cannot identify the cause of the error—if it cannot repair the source text--then, as a last resort, the procedure skips over and ignores the incorrect

code. A parsing procedure attempts to skip over as little source code as is possible in order to overlook the error and to begin parsing again. The procedure always indicates in an error message if it is about to skip over code, and it indicates, in another message, the location in the code at which it begins to parse again. In the worst case, the parsing procedure will skip to the end of the current PADL definition and begin parsing the next definition in the source file.

Chapter Four

Conclusions and Further Work

This chapter contains some criticisms of the design and implementation of the PADL parser. During the implementation of the parser, some aspects of the parser design proved to be good; these parts of the design enhanced the operation of the parser or made easier its implementation. Other aspects of the parser design were found to be less satisfactory, marring the operation of the parser or making its implementation more difficult than was necessary. Some modifications to the parser design, which should repair the design flaws, are suggested. The chapter ends with a discussion of how the PADL parser program can be put to use as a component in the FUNCHARD CAD package.

The PADL parser program is written in the CLU language. CLU has many features that helped a great deal in implementing the parser. Very helpful was the rich set of data types in CLU and the rich sets of CLU routines that access and manipulate data types. The large variety of CLU data types was useful in defining data structures for the parser output file that are both easily understandable and easily accessible. The CLU dynamic arrays, along with the iteration abstractions that yield their indices and elements, were especially useful for searching through and modifying lists of data objects. There was no use for data abstractions in the recursive-descent parsing procedures, which comprise most of the parser code, but implementing the parser support utilities – such as the lexical scanner and the message formatter – with data abstractions helped to obscure the implementation details of the support utilities from the parsing routines that used them. The language features most helpful in implementing the PADL parser were the CLU mechanisms for handling flow of control. The

CLU loop constructs and the *signal*, *exit*, and *break* mechanisms made very easy the coding of procedures that search for data, continue processing when the data or an error is found, and indicate if their searching was a success or a failure. Also helpful was the CLU method of argument passing, *call by sharing*. The call by sharing technique allows the access to a data object to be shared among many data structures and procedures. The updating of a data structure, such as the parser's symbol table, is very easy; access to the data in the symbol table is granted to parsing procedures so that the procedures can modify the symbol table data objects without ever removing them from the table.

Grouping all the parser's support utilities into a single data abstraction, the **parser** abstraction, made the implementation of the parser easier. It is certainly more convenient to pass the support utilities as a single argument, instead of as many, to all the parsing procedures. The **parser** abstraction keeps the parsing procedures and the support utilities separate and distinct. The data abstraction allows the parsing procedures to be less dependent on the details of the support utilities' implementations.

The PADL parser accomplishes all error handling within the individual parsing procedures, instead of within a single, centralized error handling routine. Handling errors in the parsing procedures results in better identification and recovery from errors than if errors were handled by a single routine. However, successfully implementing the error handling capabilities completely within the individual parsing procedures is difficult. It is difficult to foresee all the different errors that a procedure may possibly encounter. It is difficult to verify that each procedure does indeed detect and attempt to recover from every error that it may possibly encounter. It is difficult to test all the error handling capabilities of all the many parsing procedures. There seems always the possibility that an error, which was not anticipated, may exist. Such an error, if

encountered, could result in a CLU *unhandled exception* and the total failure of the parser.

The PADL parser should not attempt to accomplish all its error handling within the parsing procedures. The error handling in the parser could be implemented much more easily, be done no less well than it now is, and be more easily verified correct and complete if the parser had, in addition to the error handling capabilities of the parsing procedures, a single, general error handling routine. The parsing procedures would still check for and handle the commonly expected errors that they, with their knowledge of their constructs' correct syntax, can handle well. The central error handling routine, which could reside in the **parser** data abstraction, would handle the errors that the parsing procedures could not or did not handle. The error handling routine would recover from the errors that the parsing procedures could not identify, the errors that require skipping of source text or other general error recovery techniques. The parsing procedures would no longer need complete error handling capabilities. The error handling routine would be called to recover from the difficult or the commonplace errors. It could be called automatically when an *unhandled exception* is raised; the error handling routine could catch the errors for which the parsing procedures neglected to check.

The parser's error messages often contain insufficient information depicting an error's location. An error message shows the source file line in which an error occurs. The source file line sometimes provides a poor context from which the user must discern the location of the error, especially when the line is short or the error occurs at the edge of the line. A better error message would show, instead of a source file line, the location of the error in the context of the source file tokens that surround it. If an error message showed the five or six tokens preceding the error and the five or six tokens following the error, the

user should be able to identify more readily the location of the indicated error. Modifications to the parser's **line_buff** and **msger** abstraction s and a redefinition of the *place* object used to depict an error's context are necessary to implement this change.

The PADL parser is the first component of the FUNCHARD system to have been implemented. So that the parser could operate and be tested, it was implemented as an independent and complete system. The parser program could be used, just as it is, in the FUNCHARD system; however, some small modifications to the parser program would make it operate more naturally as a component in the FUNCHARD system.

The parser need not prompt the user for the names of PADL source files to parse, one at a time. An input program for the FUNCHARD system could receive from the user a list of all the source files that comprise a complete PADL description input. The user should be able to specify all the files in the PADL description input at one time, in one command line. The input program would automatically invoke the PADL parser to translate each of the individual source files in the input description. This would require no change to the parser program.

The PADL parser and the PADL semantic analyzer need not be separate components in the FUNCHARD system. The parser need not create an file in order to pass its translated output to the PADL semantic analyzer. The parser could be called by the semantic analyzer as a procedure. The parser could pass its output to the semantic analyzer as the output of a procedure, not requiring an output file. To accomplish this would require only a minor change in the parser program. The parser now creates its output file after it has translated the entire source file. Instead of storing the output information in a file, the parser could

simply return the output information (in the same output file format) to the calling procedure, the semantic analyzer. These minor modifications to the parser program would allow the PADL parser's incorporation as an internal component in the FUNCHARD CAD package.

Appendix A

PADL Formal Syntax

In the following BNF syntax presentation[2], pairs of curly braces in **boldface** { . . . } indicate zero or more repetitions of the material within; pairs of square brackets in **boldface** [. . .] indicate that the material within may appear at most once. PADL keywords are printed in **boldface.**

A <name> is a sequence of alphanumeric or underscore characters beginning with a letter.

<description> ::= <definition> { <definition> }

<definition> ::= | <module type def>
| <external function def>
| <data type def>

<module type def> ::= **type** <mod type header>
module (<port decl list>)
<module body>
endmod

<mod type header> ::= <mod type name> [(<param decl list>)]
= **module** (<port decl list>)
<mod type name> ::= <name>
<param decl list> ::= <decl> { ; <decl> }

<port decl list> ::= <port decl sublist> { ; <port decl sublist> }
<port decl sublist> ::= <port type> <port decl> { ; <port decl> }
<port type> ::= **inlet | outlet**

---

[2]From the PADL Reference Manual [1], Appendix II.

<port decl> ::= <port decl id list> : <data type spec>
<port decl id list> ::= <port decl id> { , <port decl id> }
<port decl id> ::= <name> [< <subscript range list> >]

<module body> ::= <structure module body> | <behavior module body>

<structure module body> ::= { <external module type decl list> }
[**submodule** <submod decl list>]
[<type external def part>]
{ <internal function def> }
<connection body>

<external module type decl list> ::= <external module type decl>
{ ; <external module type decl> }
<external module type decl> ::= **external** <mod type header>

<submod decl list> ::= <submod decl> { ; <submod decl> }
<submod decl> ::= <submod decl id list>
: <mod type name> [(<parameter list>)]
<submod decl id list> ::= <submod decl id> { , <submod decl id> }
<submod decl id> ::= <name> [{ <subscript range list> }]
<parameter list> ::= <expression>

<behavior module body> ::= [<type external def part>]
{ <internal function def> }
[<state var decl part>]
cycle
<compound action> { ; <compound action> }
endcycle

<external function def> ::= **function** <function header>
[<type external def part>]
{ <internal function def> }
<expression>
endfun

<function header> ::= <function name> (<decl> { ; <decl> }
returns <data type spec> { , <data type spec> })

<type external def part> ::= <type external def> { ; <type external def> }

<type external def> ::= <data type def> | <external function decl>
<external function decl> ::= **external** <function header>

<internal function def> ::= **function** <function header>
[<data type def part>]
{ <internal function def> }
<expression>
endfun

<data type def part> ::= <data type def> { ; <data type def> }
<data type def> ::= **type** <data type name> = <data type spec>
<data type name> ::= <name>

<data type spec> ::= <basic data type spec>
| <compound data type spec>
| <data type name>

<basic data type spec> ::= **null**
| **integer**
| **bitstr** [[<subscript range>]]

<compound data type spec> ::= **array** [<data type spec> <subscript range>]
| **record** [<field spec> { ; <field spec> }]
| **oneof** [<tag spec> { ; <tag spec> }]
[**where** <tag def> { , <tag def> }]
<field spec> ::= <field name> { , <field name> } : <data type spec>
<tag spec> ::= <tag name> { , <tag name> } [: <data type spec>]
<tag def> ::= <tag name> { , <tag name> } = <tag value>

<connection body> ::= **structure**
<conn group>
endstruct
<conn group> ::= <conn spec> { ; <conn spec> }
<conn spec> ::= <basic conn spec> | <control conn spec>

<basic conn spec> ::= <explicit conn> | <implicit conn>
<explicit conn> ::= <conn port id> -> <port list>
<implicit conn> ::= <submodule id> (<port list>)
<port list> ::= <conn port id> { , <conn port id> }
<conn port id> ::= [<submodule id> .] <port id>

<submodule id> ::= <name> [{ <subscripts> }]

<control conn spec> ::= <conditional conn> | <iterative conn>
<conditional conn> ::= **if** <condition> **then** <conn group>
{ **elseif** <condition> **then** <conn group> }
[**else** <conn group>]
endif
<iterative conn> ::= **for** <control variable> := <limit1> **to** <limit2>
<conn group>
endfor
<limit1> ::= <expression>
<limit2> ::= <expression>

<state var decl part> ::= **var** <state var decl> { ; <state var decl> }
<state var decl> ::= <decl> [:= <expression>]

<compound action> ::= <elementary action>
| <action block>
| <conditional action>
| <tagcase action>
| <iteration>
| <definition block>
<elementary action> ::= <state variable assignment>
| <state variable reference>
| <input action>
| <output action>
<state variable assignment> ::= <state var> { , <state var> }
:= <expression>
<state variable reference> ::= <state var>
<state var> ::= <name> | <state var array ref> | <state var record ref>
<state var array ref> ::= <state var>[<subscript range> { , <subscript range> }]
<state var record ref> ::= <state var> . <field name>

<input action> ::= **from** <port id list> | <tagged from>
<tagged from> ::= **tagcase** [<value name> =] <from-either list>
<tag list> : <expression>
{ <tag list> : <expression> }
[**otherwise** : <expression>]
endtag
<from-either list> ::= **from_either** <port id> , <port id list>

<tag list> ::= **tag** <port id list>

<output action> ::= **send** <expression> **at** <port id list>

<action block> ::= **begin**
<compound action> { ; <compound action> }
end

<conditional action> ::= **if** <condition> **then** <compound action>
[**else** <compound action>]
endif

<tagcase action> ::= **tagcase** [<value name> =] <expression>
<tag list> : <compound action>
{ <tag list> : <compound action> }
[otherwise : <compound action>]
endtag

<iteration> ::= **while** <condition> **do** <compound action>
| **repeat** <compound action> **until** <condition>

<definition block> ::= **let** <actdecldef part>
in <compound action>
endlet
<actdecldef part> ::= <actdecldef> { ; <actdecldef> }
<actdecldef> ::= <decl>
| <def>
| <decl> { , <decl> } = <actval>
<actval> ::= <expression> | <input operation>

<expression> ::= <level1 exp> | <expression> , <level1 exp>
<level1 exp> ::= <level2 exp> | <level1 exp> | <level2 exp>
<level2 exp> ::= <level3 exp> | <level2 exp> & <level3 exp>
<level3 exp> ::= <level4 exp> | ~ <level4 exp>
<level4 exp> ::= <level5 exp> | <level4 exp> <relational op> <level5 exp>
<level5 exp> ::= <level6 exp> | <level5 exp> || <level6 exp>
<level6 exp> ::= <level7 exp> | <level6 exp> <adding op> <level7 exp>
<level7 exp> ::= <level8 exp> | <level7 exp> <multiplying op> <level8 exp>
<level8 exp> ::= <primary> | <unary op> <primary>

<relational op> ::= < | <= | > | >= | == | ~=
<adding op> ::= + | -
<multiplying op> ::= * | /
<unary op> ::= + | -

<primary> ::= <constant>
| <value name>
| (<expression>)
| <function invocation>
| <array ref> | <array generator>
| <record ref> | <record generator>
| <oneof test> | <oneof generator>
| <prefix operation>
| <conditional exp>
| <letin exp>
| <tagcase exp>
| <forall exp>

<constant> ::= **nil**
| true | false
| <integer number>
| <bit string constant>

<function invocation> ::= <function name> (<expression>)

<array ref> ::= <primary> [<subscripts>]
<array generator> ::= <primary> [<subscript range list>]

<record ref> ::= <primary> . <field name>
<record generator> ::= **record** [<field name> : <expression>
{ ; <field name> : <expression> }]

<oneof test> ::= **is** <tag name> (<expression>)
<oneof generator> ::= **make** <data type spec> [<tag name> : <expression>]

<prefix operation> ::= <prefix operator> (<expression>)
<prefix operator> ::= **abs** | **exp** | **mod** | **shifl** | **shifr** | **rotl** | **rotr** | **bitstr** | **integer**

<conditional exp> ::= **if** <condition> **then** <expression>
{ **elseif** <condition> **then** <expression> }

else <expression>
endif

<letin exp> ::= **let** <decldef part>
in <expression>
endlet

<tagcase exp> ::= **tagcase** [<value name> =] <expression>
<tag list> : <expression>
{ <tag list> : <expression> }
[**otherwise** : <expression>]
endtag

<tag list> ::= **tag** <tag> { , <tag> }
<tag> ::= <tag value> | <tag name>
<tag value> ::= <bit string constant> | <integer number>

<forall exp> ::= **forall** <value name> **in** [<expression>]
{ , <value name> **in** [<expression>] }
[<decldef part>]
<forall body part>
{ <forall body part> }
endall
<forall body part> ::= **construct** <expression>
| **eval** <forall op> <expression>
<forall op> ::= **plus** | **times** | **min** | **max** | **or** | **and**

<bit string constant> ::= '<bit string> | #<octal string> | @<hexadecimal string>
<bit string> ::= <binary char> { <binary char> }
<octal string> ::= <octal char> { <octal char> }
<hexadecimal string> ::= <hex char> { <hex char> }
<binary char> ::= ? | <binary digit>
<octal char> ::= ? | <octal digit>
<hex char> ::= ? | <hexadecimal digit>

<condition> ::= <expression>

<decldef part> ::= <decldef> { ; <decldef> }
<decldef> ::= <decl> | <def> | <decl> { , <decl> } = <expression>
<def> ::= <name> { , <name> } = <expression>

<decl> ::= <name> { , <name> } : <data type spec>

<field name> ::= <name>
<function name> ::= <name>
<tag name> ::= <name>
<value name> ::= <name>

<port id list> ::= <port id> { , <port id> }
<port id> ::= <name> [< <subscripts> >]

<subscript range list> ::= <subscript range> { , <subscript range> }

<subscript range> ::= <expression> : <expression>

<subscripts> ::= <expression>

Appendix B

The Parser Output File

The output file of the PADL parser is written using a CLU *istream*. It must be read from an *istream*, using the CLU *decode* routines for the data types of its contents.

4.1 Contents

The following is a list of the output file contents. The contents are written to the file in this order; they must be read from the file in this same order.

error: **bool**,
source_name: file_name,
source_cdate: date,
name_list: **array**[**string**],
typespec_list: **array**[typespec],
exttype_list: **array**[**int**],
extfun_list: **array**[extfun],
modtype_list: **array**[modtype]

error is a flag which is **true** if the parser detected a syntactic or semantic error in the source file.

source_name and *source_cdate* are the name and creation date of the PADL source file from which the output file was produced.

The *name_list* is a list of the name identifiers used in the source file.

The *typespec_list* is a list of the data type specifications used in the source file for PADL data elements.

The *exttype_list*, *extfun_list*, and *modtype_list* are the translations of the data type, function, and module type definitions in the PADL source file. The translations are parse tree representations of the PADL definitions which comprise the PADL description.

4.2 Data Structures

A listing of the actual CLU equates used to define the data structures of the PADL parser output file follows. output file. A more detailed explanation of these data structures—their contents and their meanings—can be found in a separate document [4].

```
%% The Name List %%

name_list = array[ string ]


%% The Type Specification List %%

typespec_list = array[ typespec ]

typespec = record[
                name: int,
                undefined: bool,
                dtype: dtype
                  ]

dtype = oneof[
            terror: null,
            tnotype: null,
            texttype: int,
            tnull: null,
            tbitstr: range,
            tinteger: null,
            tarray: arrayspec,
            trecord: recordspec,
```

```
                toneof: oneofspec
                  ]

arrayspec = record[
                  eltype: int,
                  bounds: range
                    ]

recordspec = record[
                   fldnames: array[ int ],
                   fldtypes: array[ int ]
                     ]

oneofspec = record[
                   tagnames: array[ int ],
                   tagcodes: tagcodes,
                   tagtypes: array[ int ]
                     ]

tagcodes = oneof[
                  none: null,
                  icodes: array[ int ],
                  bcodes: array[ string ]
                    ]

range = record[
              low: expression,
              high: expression
                ]


%% The External Data Type List %%

exttype_list = array[ int ]


%% The External Function List %%

extfun_list = array[ extfun ]
```

```
extfun = record[
            extfun_decl_list: array[ extfun_decl ],
            intfun_list: array[ funbody ],
            body: funbody
              ]

extfun_decl = record[
                name: int,
                arg_types: array[ int ],
                ret_types: array[ int ]
                  ]

funbody = record[
            name: int,
            lineno: int,
            blockno: int,
            arg_names: array[ int ],
            arg_types: array[ int ],
            ret_types: array[ int ],
            funval: array[ expression ]
              ]


%% The Module Type List %%

modtype_list = array[ modtype ]

modtype = oneof[
            bmod: bmodtype,
            smod: smodtype
              ]

bmodtype = record[
            header: modheader,
            extfun_decl_list: array[ extfun_decl ],
            intfun_list: array[ funbody ],
            stvar_names: array[ int ],
            stvar_types: array[ int ],
            stvar_values: array[ expression ],
            action_list: array[ action ]
```

```
                    ]

smodtype = record[
                header: modheader,
                submod_decl_list: array[ submod_decl ],
                extfun_decl_list: array[ extfun_decl ],
                intfun_list: array[ funbody ],
                connection_list: array[ connection ]
                    ]

modheader = record[
                    name: int,
                    lineno: int,
                    blockno: int,
                    param_names: array[ int ],
                    param_types: array[ int ],
                    inport_list: array[ port_decl ],
                    outport_list: array[ port_decl ]
                     ]

port_decl = record[
                name: int,
                dtype: int,
                subs: array[ sub ]
                    ]

submod_decl = record[
                    name: int,
                    subs: array[ sub ],
                    modtype_name: int,
                    params: array[ expression ]
                       ]


%% PADL Expressions %%

exp = oneof[
            none: null,
            arity_place: null,
            nilcon: null,
```

```
                btscon: string,
                intcon: int,
                valname_ref: valname_ref,
                array_create: array_create,
                array_select: array_select,
                record_create: record_create,
                record_select: record_select,
                oneof_create: oneof_create,
                oneof_test: oneof_test,
                operation: operation,
                ifthen: ifthen,
                letin: letin,
                tcase: tcase,
                forall: forall,
                intfun_call: intfun_call,
                extfun_call: extfun_call
                  ]

valname_ref = record[
                    blockno: int,
                    valno: int,
                    ref_name: int,
                    ref_type: int,
                    stvar: bool
                      ]

array_create = record[
                    start_idx: expression,
                    args: array[ expression ],
                    arg_type: int,
                    ret_type: int
                      ]

array_select = record[
                    arg: expression,
                    arg_type: int,
                    idxs: array[ sub ],
                    ret_type: int
                      ]
```

```
sub = oneof[
        single: expression,
        pair: range
          ]

record_create = record[
                  args: array[ expression ],
                  arg_types: array[ int ],
                  ret_type: int
                    ]

record_select = record[
                  arg: expression,
                  arg_type: int,
                  fields: array[ int ],
                  ret_type: int
                    ]

oneof_create = record[
                  tagno: int,
                  value: expression,
                  ret_type: int
                    ]

oneof_test = record[
                arg: expression,
                tagno: int
                  ]

operation = record[
                operator: op,
                arg_types: array[ int ],
                args: array[ expression ],
                ret_type: int
                  ]

ifthen = record[
            tests: array[ expression ],
            arms: array[ array[ expression ] ],
            ret_types: array[ int ]
```

```
            ]

letin = record[
            blockno: int,
            lcl_names: array[ int ],
            lcl_types: array[ int ],
            lcl_values: array[ expression ],
            ret_types: array[ int ],
            ret_values: array[ expression ]
              ]

tcase = record[
            blockno: int,
            arg: expression,
            arg_type: int,
            lcl_names: array[ int ],
            tagstype: tagstype,
            arms: array[ array[ expression ] ],
            arm_types: array[ int ],
            armtab: array[ int ]
              ]

tagstype = oneof[
              names: null,
              bts: array[ int ],
              ints: null
                ]

forall = record[
            blockno: int,
            idx_limits: array[ range ],
            lcl_names: array[ int ],
            lcl_types: array[ int ],
            lcl_values: array[ expression ],
            clauses: array[ array[ expression ] ],
            eval_ops: array[ eval_op ],
            ret_types: array[ int ]
              ]

eval_op = oneof[
```

```
                construct,
                plus,
                times,
                min,
                max,
                or,
                and: null
                  ]

intfun_call = record[
                intfunno: int,
                args: array[ expression ],
                ret_types: array[ int ]
                  ]

extfun_call = record[
                extfunno: int,
                args: array[ expression ],
                ret_types: array[ int ]
                  ]

op = oneof[
          bts_and,
          bts_or,
          bts_not,
          bts_eq,
          bts_noteq,
          bts_concat,
          bts_shifl,
          bts_shifr,
          bts_rotl,
          bts_rotr,
          bts_substr,
          int_unminus,
          int_unplus,
          int_plus,
          int_minus,
          int_times,
          int_div,
          int_mod,
```

```
            int_exp,
            int_eq,
            int_noteq,
            int_grtr,
            int_less,
            int_grtreq,
            int_lesseq,
            bitsr_to_int,
            int_to_bts: null
              ]

%% PADL Actions %%

act = oneof[
            stvar_assgn: stvar_assgn,
            from: from,
            fromeither: fromeither,
            send: send,
            block_act: array[ action ],
            ifthen_act: ifthen_act,
            tcase_act: tcase_act,
            while_act: while_act,
            repeat_act: repeat_act,
            letin_act: letin_act
              ]

stvar_assgn = record[
                  stvar_refs: array[ stvar_ref ],
                  assgn_types: array[ int ],
                  assgn_values: array[ expression ]
                    ]

stvar_ref = oneof[
                simple_ref: stv_ref,
                array_ref: stvarray_ref,
                record_ref: stvrec_ref
                  ]

stv_ref = record[
```

```
                ref_name: int,
                ref_type: int,
                blockno: int,
                stvno: int
                   ]

stvarray_ref = record[
                      ref_arg: stv_ref,
                      idxs: array[ sub ],
                      ref_type: int
                         ]

stvrec_ref = record[
                    ref_arg: stv_ref,
                    fields: array[ int ],
                    ref_type: int
                       ]

from = record[
              stvar_refs: array[ stvar_ref ],
              ports: array[ port_id ]
                 ]

port_id = record[
                 port: int,
                 subs: array[ sub ]
                    ]

fromeither = record[
                      blockno: int,
                      stvar_refs: array[ stvar_ref ],
                      ports: array[ port_id ],
                      lcl_names: array[ int ],
                      arms: array[ array[ expression ] ],
                      arm_types: array[ int ],
                      armtab: array[ int ]
                         ]

send = record[
              value: expression,
```

```
            ports: array[ port_id ]
              ]

block_act = array[ action ]

ifthen_act = record[
                tests: array[ expression ],
                arms: array[ action ]
                  ]

tcase_act = record[
                blockno: int,
                arg: expression,
                arg_type: int,
                lcl_names: array[ int ],
                tagstype: tagstype,
                arms: array[ action ],
                armtab: array[ int ]
                  ]

while_act = record[
                act: action,
                test: expression
                  ]

repeat_act = record[
                act: action,
                test: expression
                  ]

letin_act = record[
                blockno: int,
                lcl_names: array[ int ],
                lcl_types: array[ int ],
                lcl_values: array[ expression ],
                act: action
                  ]


%% PADL Connection Specifications %%
```

```
conn = oneof[
            expl_conn: expl_conn,
            impl_conn: impl_conn,
            ifthen_conn: ifthen_conn,
            iter_conn: iter_conn
              ]

expl_conn = record[
                outport: connport_id,
                inports: array[ connport_id ]
                  ]

connport_id = record[
                  submod: submod_id,
                  port_name: int,
                  subs: array[ sub ]
                    ]

submod_id = record[
                  submodno: int,
                  subs: array[ sub ]
                    ]

impl_conn = record[
                 submod: submod_id,
                 ports: array[ connport_id ]
                   ]

ifthen_conn = record[
                   tests: array[ expression ],
                   arms: array[ array[ connection ] ]
                     ]

iter_conn = record[
                 blockno: int,
                 cvar_name: int,
                 limits: range,
                 conns: array[ connection ]
                   ]
```

4.2.1 Expressions, Actions, and Connections

The syntax definitions for PADL expressions, actions, and connection specifications are self-recursive. CLU equates alone cannot represent recursive data structures; they cannot represent these PADL constructs. The parser output file representations for these constructs are CLU data abstraction clusters. The data abstractions for expressions, actions, and connections are implemented with CLU records. The data in an *expression*, *action*, or *connection* object can be accessed as if the object were simply a record. The abstraction clusters provide *get* and *set* routines for each of their record fields, and also *encode* and *decode* routines for the entire abstract data types.

The following shows the structures of the *expression*, *action*, and *connection* pseudo-records.

```
expression = record[
                lineno: int,
                data: exp,
                typ: int,
                pure: bool
                  ]

action = record[
            lineno: int,
            data: act
              ]

connection = record[
                lineno: int,
                data: conn
                  ]
```

References

[1] Leung, Clement K.C., Lim, William Y-P.
PADL – A Packet Architecture Description Language. A Preliminary Reference Manual.
Technical Report, Laboratory for Computer Science, M.I.T., Cambridge, MA, September, 1983.
(To be published)

[2] Lim, Willie Y-P., Leung, Clement K.C.
PADL – A Packet Architecture Description Language.
Computation Structures Group Memo 221, Laboratory for Computer Science, M.I.T., Cambridge, MA, October, 1982.
(Presented at the *Sixth International Symposium on Computer Hardware Description Languages and their Applications*, Carnegie-Mellon University, Pittsburgh, PA, May 23-25, 1983)

[3] Lim, Willie Y-P., Wanuga, Thomas S.
FUNCHARD – An Expert System for Translating Functional Specifications to Hardware Structures.
Computation Structures Group, Laboratory for Computer Science, M.I.T., Cambridge MA. (Unpublished)

[4] Holderle, James E.
PADL Parser Output: The ***bnp*** *File.*
Computation Structures Group, Laboratory for Computer Science, M.I.T., Cambridge MA. (Unpublished)

[5] Liskov, Barbara, *et. al.*.
CLU Reference Manual.
Technical Report MIT/LCS/TR-225, Laboratory for Computer Science, M.I.T., Cambridge, MA, October, 1979.

[6] Ackerman, William B., Dennis, Jack B.
VAL – A Value-Oriented Algorithmic Language: Preliminary Reference Manual.
Technical Report MIT/LCS/TR-218, Laboratory for Computer Science, M.I.T., Cambridge, MA, June, 1979.