TOOLS FOR TESTING DENOTATIONAL SEMANTIC DEFINITIONS OF
PROGRAMMING LANGUAGES. (U) UNIVERSITY OF SOUTHERN
CALIFORNIA MARINA DEL REY INFORMATION S.
UNCLASSIFIED V KINI ET AL. MAY 83 ISI/RR-83-112
Tools for Testing Denotational Semantic Definitions of Programming Languages
Tools for Testing Denotational Semantic Definitions of Programming Languages

Vittal Kini, David F. Martin, and Allen Stoughton

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

This document is approved for public release and sale: distribution is unlimited.

Ada, denotational semantics, semantic interpreter, software tools
20. ABSTRACT (continued)

The Department of Defense commissioned the design and implementation of the Ada programming language with the intention of requiring most future military systems to be programmed in Ada. To ensure the quality of systems written in Ada, it is necessary that Ada be precisely understood by both its users and implementers. To this end a denotational formal semantic definition (FSD) of Ada was developed at the Institut National de Recherche en Informatique et en Automatique (INRIA), in France. Its intent is to provide the precise meaning of the language and its constructs via the mathematical formalism which underlies the denotational semantic descriptive technique. Due to the complexity of Ada, however, and despite the power and elegance of the denotational semantics method, the FSD itself is quite large: both the static (compile-time) and dynamic (run-time) phases of the definition consist of hundreds of mutually recursive functions. As a result of this inherent complexity, it is difficult for a person to understand the definition or to attempt symbolic execution of the definition without machine assistance. This report describes the work of the ISI Formal Semantics project in developing and constructing tools to aid the understanding and validation of the Ada FSD. First we briefly describe the INRIA meta-language, AFDL, and the extensions we were forced to make to it. Next we describe the various tools we have built and their application to the interpreting of the FSD. Finally, we describe the outcome of the project. The appendices contain an informal specification of our enhanced version of AFDL (AFDL+), a definition of the toy programming language TINY in AFDL+, and a transcript of an example use of our tools to process the TINY definition.
Tools for Testing Denotational Semantic Definitions of Programming Languages
# CONTENTS

1. Testing the INRIA Ada Formal Definition: The ISI Formal Semantics Project .................................................. 1  
   1.1 Introduction ......................................................................................................................... 1  
   1.2 Ada Formal Semantic Definition ...................................................................................... 2  
   1.3 Mechanical Interpretation of the Ada FSD ..................................................................... 3  
   1.4 Status as of June 1982 ................................................................................................. 8  

2. A Description of AFDL +: The Ada FSD Metalanguage .................................................................................. 10  
   2.1 Overview .......................................................................................................................... 10  
   2.2 Lexical Elements .............................................................................................................. 11  
   2.3 Declarations and Types ................................................................................................. 12  
   2.4 Expressions .................................................................................................................... 19  
   2.5 Functions ....................................................................................................................... 25  
   2.6 Packages ........................................................................................................................ 26  
   2.7 Program Structure ........................................................................................................... 26  

3. AFDL + Transcripts .......................................................................................................................... 27  
   I. ISI Extensions to AFDL ........................................................................................................ 60  
   II. Denotational Semantic Definition of TINY in AFDL + ................................................ 62  
   III. AFDL + Abstract Syntax ................................................................................................. 66  
   IV. AFDL + Concrete and Abstract Syntax ......................................................................... 69  

References ............................................................................................................................................. 77
1. TESTING THE INRIA ADA FORMAL DEFINITION: THE ISI FORMAL SEMANTICS PROJECT

1.1 INTRODUCTION

The design and implementation of the Ada [1] programming language were commissioned by DoD with the intention of requiring most future military systems to be programmed in Ada. It is therefore necessary that Ada be precisely understood by both its users and implementers, in order to ensure the quality of systems written in Ada. In particular, since DoD must control Ada compiler implementations, a precise, well-structured, and validated formal definition of Ada can provide one of the principal standards to which these implementations must adhere. Beyond such considerations pertaining to particular programming languages such as Ada, good generic research toward the design and implementation of tools and methodologies for supporting the development of precise, readable, and accurate formal definitions has considerable relevance to the broader goals of understanding large programs and verifying their correctness.

A denotational formal semantic definition (FSD) of Ada has been developed at INRIA [10]. Due to the complexity of Ada, and despite the power and elegance of the denotational semantics method, the FSD itself is quite large: both the static (compile-time) and dynamic (run-time) phases of the definition consist of hundreds of mutually recursive functions. As a result of this inherent complexity, it is difficult for a human to understand the definition. One approach to understanding a denotational definition is to symbolically execute the definition on specific example programs. Attempting to do this without machine assistance will likely result in a great many errors and is, in a practical sense, impossible. Unaided human application of this FSD to understanding Ada programs is at best an arduous task.

It is therefore imperative to construct appropriate tools to aid the understanding and validation of the Ada FSD. Such tools can be used in two ways. Initially, Ada test cases whose semantics are well understood can be used to test the correctness of the FSD. Subsequently, after confidence in the correctness of the Ada FSD has increased, the tools can be used to answer very specific questions about specific parts of the FSD as they relate to example Ada programs whose semantics are not readily apparent. This report describes work done at USC-Information Sciences Institute in constructing tools that may be used in these ways to exercise and validate the INRIA Ada FSD.

The Ada FSD is written in a typed lambda calculus expressed in an "Ada-like" syntax; we shall henceforth call this language AFDL, an acronym for Ada Formal Definition Language. We have written tools to

- translate the functions and data types of the Ada FSD into an equivalent directly executable intermediate language (AFDL-IL),
- transform candidate Ada test programs (such as the Softech compiler test cases [8]) into corresponding abstract syntax trees, and
- apply the translated FSD to the abstract syntax trees to obtain via interpretation the static and dynamic semantics of the corresponding programs.

The semantics thus obtained can be compared to the expected meaning. In addition we have built tools to generate useful items such as cross reference listings of the FSD's components.
In this report, we describe in more detail our approach to validating the Ada FSD. First, we briefly describe the INRIA meta-language and the extensions we were forced to make to it, and give an overview of the structure of the INRIA Ada FSD. Next, we describe the various tools we have built and their application to the FSD. And finally, we describe the current state of our project and its expected outcome.

1.2 ADA FORMAL SEMANTIC DEFINITION

1.2.1 The Meta-Language of the Ada FSD

AFDL, the meta-language in which the FSD is written, is an applicative language with an Ada-like syntax. The language contains function, block, conditional, and case statements, simple expressions, and packages (for modularity and information hiding) as in Ada. AFDL's basic data types are the integers and the booleans, and its data type constructors are enumerated types and (unlike Ada) function types. Conspicuously absent from AFDL are the sum (union), product (record), and sequence (array) types; these types are basic to the denotational semantics method. As a result, the data types upon which the FSD is based are only defined informally in plain language, and are not formally defined in AFDL (or any other language). The absence of formal explication of the entire base-level of the definition is one of the major impediments to the human reader who wishes to understand the formal definition (the task is akin to trying to understand a large software system in which the data type declarations were only cursorily outlined in English). This deficiency must be remedied before the FSD can be tested.

Our approach to this problem was to extend AFDL, in an upward compatible way, to include sum, product and sequence types, along with their associated operations. This approach provides a meta-language capable of conveying the entire definition in a formal manner, and thus facilitates both human understanding and machine execution of the definition. We call this extension AFDL+. Further details of AFDL+ are provided in Appendix I. Appendix II of the report contains a definition of Gordon's example programming language "TINY" [9]. The definition is essentially a transliteration of the continuation-style semantic definition of TINY that Gordon provides. In this definition, all of the static and semantic domains are defined in terms of the basic types INTEGER and BOOLEAN and the various domain constructors.

1.2.2 Structure of the Ada FSD

The Ada FSD basically consists of a collection of mutually recursive functions together with a repertoire of intrinsic basic data types. The Ada Reference Manual [1] is divided into a number of chapters, each devoted to a specific aspect or component of the language. None of the FSD appears in this manual. The Ada FSD ([10] and later versions), however, is organized by "folding" it into the Reference Manual so that each chapter contains the pertinent components of the Ada FSD. The intrinsic FSD data types and operations on them are intended to be described in Appendices to the Ada FSD in order to separate these base-level concerns from the rest of the FSD.

From an operational point of view, the Ada FSD is organized into three "phases", one of which is syntactic and the others semantic. The syntactic phase establishes a relationship between the concrete and abstract syntax of Ada by providing a specification of both the concrete and abstract
syntactic domains together with a (constructive) mapping from the former to the latter. In practice, this mapping is implemented as a parse-driven construction of Ada abstract syntax trees from corresponding Ada program strings. There are two semantic phases which process abstract syntax trees. The first, called static semantics, performs what are generally considered to be "compile-time" functions such as static type-checking and overloading resolution. This phase produces what we term an annotated abstract syntax tree which is a modified form of the tree input to the phase. The annotations consist of error messages and static environment information gathered by the phase such as the overloading resolutions, visibility calculations, normalizations, etc. The final semantic phase, called dynamic semantics, determines the "run-time" semantics (the meaning of procedures, expressions, etc.) of error-free annotated abstract syntax trees output from the static semantics phase.

1.2.3 Status of the Ada FSD -- June 1982

At this time, the Ada FSD is incomplete in that (excluding the semantics of Tasking in Ada) many of the semantic functions in the chapters of the FSD document are either missing or contain errors (both type and logical), and the data types and functions in appendices of the document are largely unspecified. The functions missing from the FSD have been identified, and type errors have been detected by the use of type-checking tools developed at ISI. The burden of defining these missing types and functions and of correcting these errors necessarily falls upon INRIA.

In the FSD document, the concrete and abstract syntaxes of Ada are defined, the latter less explicitly than the former. In fact, there exist two abstract syntaxes: a post-parse abstract syntax (which is input to the static semantics phase) and a post-static-semantics abstract syntax (which is input to the dynamic semantics phase). The correspondence between the (post-parse) abstract syntax and the concrete syntax in the document is implicit; it must be given explicitly or else the reader must be given enough information to deduce the exact correspondence, as this is necessary for a detailed understanding of the FSD. However, INRIA has supplied an LR(1) syntax for a superset of Ada, together with a correspondence between it and the Ada abstract syntax.

1.3 MECHANICAL INTERPRETATION OF THE ADA FSD

In this section we shall describe the processes whereby the Ada FSD is translated into suitable intermediate forms that may then be interpreted. The interpretive execution of the FSD will allow the testing and validation of the FSD and, subsequently, the accurate determination of the semantics of given example Ada programs. Figures 1, 2, 3, and 4 illustrate these processes.

The boxes in these figures represent abstract machines that accept inputs and produce outputs in certain forms. Boxes subdivided vertically into two compartments represent abstract machines that are constructed by loading what is represented by the upper compartment into the abstract machine represented by the lower compartment. Compartments may themselves be vertically subdivided into compartments in a hierarchical manner (see Figure 4 for example). Numbers above the top left corner of a box uniquely identify an abstract machine and two boxes labeled with the same number therefore represent the same abstract machine.
Parser Generation

1

<table>
<thead>
<tr>
<th>Parser</th>
<th>Generator</th>
<th>&quot;LR&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFDL LR(1) Syntax</td>
<td>----</td>
<td>----</td>
</tr>
</tbody>
</table>

1

<table>
<thead>
<tr>
<th>Parser</th>
<th>Generator</th>
<th>&quot;LR&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada LR(1) Syntax</td>
<td>----</td>
<td>----</td>
</tr>
</tbody>
</table>

Figure 1

Processing FSD Semantic Functions

A (from Fig. 1)

<table>
<thead>
<tr>
<th>FSD</th>
<th>Static</th>
<th>---</th>
<th>+---</th>
<th>--&lt;&gt;</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>--- (to C; Fig. 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v</td>
<td>Fns.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AFDL</td>
<td>Syntax</td>
<td></td>
<td>Type</td>
<td>Checked</td>
<td>AFDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source</td>
<td>Tables</td>
<td></td>
<td>Chk'g.+</td>
<td>Checked</td>
<td>AFDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>T.D.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FSD</td>
<td>Parser</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
<td>---</td>
<td>Fns.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td></td>
<td>AFDL Syntax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td></td>
<td>Errors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td></td>
<td>AFDL Static</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td></td>
<td>Errors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key: T.D. Parser = Table Driven Parser

Figure 2
Static Semantics (PASS 1)

\[
\begin{array}{c}
\text{Ada Program} \\ \downarrow \text{T.D.} \\
\text{Syntax} \\ \downarrow \text{Parser} \\
\text{B (from Fig. 1)} \\ \downarrow \text{Syntax} \\
\text{Tables} \\
\text{Ada Program} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Ada Program} \\
\downarrow \text{AFDL-IL} \\
\text{Abstract Syntax} \\
\downarrow \text{Interpreter} \\
\text{C (from Fig 2)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Ada Program} \\
\downarrow \text{AFDL-IL} \\
\text{Annotated (to E; Fig. 4)} \\
\end{array}
\]

Key: T.D. Parser = Table Driven Parser

Figure 3

Dynamic Semantics (PASS 2)

\[
\begin{array}{c}
\text{Ada Program} \\
\downarrow \text{Input Data} \\
\text{B} \\
\downarrow \text{Dynamic} \\
\text{AFDL-IL} \\
\downarrow \text{Interpreter} \\
\text{E (from Fig. 3)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{E (from Fig. 3)} \\
\downarrow \text{Output Data} \\
\text{Ada Program} \\
\downarrow \text{AFDL-IL} \\
\text{D (from Fig. 2)} \\
\end{array}
\]

Figure 4
1.3.1 Generation of Parsers

To generate parsers for the Ada and AFDL languages we use the LR program [12] which accepts an LR(1) syntax for a language and outputs syntax tables. These tables are used to control a table-driven LR parser. Figure 1 shows this process for Ada and AFDL. Abstract Machine 1 is the LR program referred to above. The table-driven parser (bottom half, Abstract Machines 2 and 6) was constructed in Interlisp.

1.3.2 Processing of FSD Semantic Functions

The static and dynamic semantic functions of the FSD, written in AFDL, are translated into a directly executable intermediate language, AFDL-IL, by the process depicted in Figure 2. Abstract Machine 2 parses the AFDL text of the functions and produces the AFDL abstract syntax (AFDL-AS) form of those functions (the AFDL text is first extracted from the surrounding prose in the FSD source document, using a SNOBOL program). Errors discovered at this stage include AFDL syntax errors arising from misspelled or missing keywords or delimiters, or from improperly constructed syntactic forms. The syntax tables generated by the LR program in the process of Figure 1 are loaded into the table-driven LR(1) parser to yield Abstract Machine 2.

The output of Abstract Machine 2 is then fed to Abstract Machine 3 which checks for incorrect AFDL data type usage and resolves any overloading of function names. Errors trapped at this stage consist of improper type usage, unresolvable function name overloading due to ambiguity, missing declarations, misspelled declaration names leading to the appearance of missing declarations, etc. The output of Abstract Machine 3 is the AFDL-AS form of the static and dynamic semantic functions in which such errors have been eliminated. The actual output of the type-checker/overloading-resolver is a symbol table that relates all symbols in the global scope of the definition to their type-checked values. Thus, for example, the value of a symbol representing a typename will be the definition of that type and the value of a symbol representing a function will be the type of that function along with the abstract syntactic form of the function body.

For efficiency, the abstract syntax tree forms of the semantic functions are not directly executed during the interpretation process. A compiled form is used instead. Abstract Machine 4 compiles the AFDL-AS functions, producing instruction sequences for the AFDL-IL interpreters depicted in Abstract Machines 7 and 8. The AFDL-IL interpreter is a virtual stack machine implemented in Interlisp with a retention strategy for storage to model the static binding of variables which is possible in AFDL.

1.3.3 Cross Reference Facility

Abstract Machine 5 in Figure 2 represents the cross reference facility. This program accepts the AFDL-AS output from Abstract Machine 3 and produces a listing of attributes for each symbol in the global scope of the FSD. Thus, for instance, if a symbol represents a semantic function, the attributes include a list of functions it calls and a list of functions that call it. These cross reference lists are a useful tool in checking the FSD for irregularities.
1.3.4 Static Semantics, PASS 1

The Ada FSD is constructed to provide the semantics of an Ada program in two passes that evaluate, respectively, the static and dynamic semantics of the program. The process of Figure 3 depicts the first or static semantics phase. The Ada programs whose semantics are to be evaluated are first rendered into an abstract syntactic form by an LR(1) Ada parser (Abstract Machine 6). This abstract machine is constructed by loading the syntax tables for Ada, produced by the LR program, into the table-driven parser mentioned in Section 1.3.1.

The Ada program abstract syntax is fed into Abstract Machine 7 which does the actual static semantic evaluation. Abstract Machine 7 is constructed by loading the compiled AFDL-IL form (symbol table) of the static semantic functions of the FSD (output from Abstract Machine 4, Figure 2) into an interpreter for AFDL-IL. As mentioned earlier, the interpreter is a stack machine programmed in Interlisp. Abstract Machine 7 is, therefore, the static semantics evaluator and embodies the corresponding part of the FSD.

The output of Abstract Machine 7 is an annotated abstract syntactic representation of the Ada program that was input to the process of Figure 3. The annotations correspond to the error messages and static environment information that were obtained from the program and checked for correctness by the static semantic functions of the FSD. By using this annotated abstract syntax representation, the dynamic semantic functions in PASS 2 need not re-analyze the Ada program to obtain static environment information.

During the running of Abstract Machine 7, AFDL dynamic (run-time) errors may occur (e.g. selection of a non-existent element of a sequence type (array)). Such errors are defects in the FSD and must be corrected. In addition, various errors may also be detected by the FSD itself and recorded as annotations in the Ada abstract syntax tree it produces. For instance, unresolvable overloading of Ada function or operator names, ambiguities due to importation of declarations by means of use clauses, improper data-type usage, etc., are errors that may be statically determined to exist in a program. The flagging of such errors, however, may signify different things depending on the intended use of Abstract Machine 7. In initial phases when the FSD is being validated, it is supposed that the semantics of the test-case input Ada programs are well understood and are known to conform to program specifications. In such cases, the errors flagged by Abstract Machine 7 will in all likelihood be caused by errors in the FSD static semantic functions themselves. In latter phases, subsequent to the validation of the FSD, such errors flagged by Abstract Machine 7 will point to errors in the Ada programs themselves. In such cases, the FSD static semantic functions will have fulfilled their purpose.

1.3.5 Dynamic Semantics, PASS 2

The second pass of the Ada FSD, the dynamic semantics phase, is depicted in the process of Figure 4. It is seen from the figure that Abstract Machine 8 is constructed in a manner similar to that of Abstract Machine 7. This is done by loading the compiled AFDL-IL (symbol table) form of the FSD dynamic semantic functions (output of Abstract Machine 4 in Figure 2) into the AFDL-IL interpreter. The result is an abstract machine with the capability to evaluate Ada program annotated abstract syntax trees.
The dynamic semantics interpreter (Abstract Machine 9) is obtained by loading Abstract Machine 8 with the annotated abstract syntax representation of the Ada program that is the output of the static semantics evaluator (Abstract Machine 7). Machine 9 is an executable object that embodies the semantics of these Ada programs. It accepts representations of the input data that the Ada programs would have accepted and through a process of interpretation produces the output data of those programs (assuming they terminate). In addition to viewing the output data produced by Ada test programs, it is also possible to view the internal computation states of those programs. Both the external and internal behavior can be used in judging the correctness of the FSD and Ada programs.

In a manner similar to that described in Section 1.3.4, AFDL dynamic (run-time) errors can occur during the execution of Abstract Machine 9. These are FSD errors. In addition, the FSD dynamic semantic functions themselves may detect run-time errors in the Ada test programs. These errors are indicated as part of the output data of those programs, and may represent errors either in the dynamic functions of the FSD or in the Ada programs themselves. Furthermore, the program output data produced by Abstract Machine 9 may be examined to determine whether it is as expected. Again, in the case of test-case Ada programs with well understood semantics, incorrect output data is symptomatic of errors in the FSD which will require identification and correction.

1.3.6 Summary

It may be observed from the foregoing description that the processes depicted in Figures 1 and 2 are performed once for each version of the Ada FSD. (Except for the process in Figure 1 which generates the syntax tables for the LR(1) Ada grammar. This is done precisely once, unless the grammar for Ada changes, a much less likely event.) Here, a new version of the Ada FSD is created each time errors detected in the FSD are corrected. Therefore, these processes are "constructor processes" in that they are used to construct the Ada FSD validation tools. When errors in the FSD are detected during the running of these constructor processes, they must be terminated prematurely, a new version of the FSD generated, and these processes restarted.

The processes depicted in Figures 3 and 4, on the other hand, are run for each test-case Ada program whose semantics is to be evaluated, or whose known semantics is to be used to validate the FSD. As such, these processes may be viewed as "validation processes".

In summary, the processes of Figures 1 and 2 are used to parse, type-check and prepare intermediate forms of the FSD static and dynamic semantic functions. The checked intermediate forms are then loaded into the AFDL-IL interpreter to produce the static and dynamic semantics interpreters. These interpreters are used in the processes of Figures 3 and 4 to evaluate the semantics of test-case Ada programs and validate the FSD.

1.4 STATUS AS OF JUNE 1982

The software tools we have described in this report have been constructed and are fully operational at USC-Information Sciences Institute. An exception is the user interface to the AFDL-IL interpreter which must perform remain in a rudimentary state until experience with exercising the complete INRIA Ada FSD is gained. For the moment we have the capability to set conditional breakpoints upon entries to and/or exits from semantic functions and also at the level of individual instructions in the compiled forms of the semantic functions. The full power of Interlisp is available at a breakpoint and thus it is possible to observe the static and dynamic chains, variable bindings and continuations in the
interpreter. This ability to intervene interactively in the interpretation is a feature that distinguishes our work from that of Mosses [11].

An extensive list of type errors generated by the type checker (Abstract Machine 3), and cross reference listings have been supplied to INRIA for the entire FSD. Unfortunately, because the FSD is currently incomplete, it has not yet been possible to test the static and dynamic interpreters on the FSD, or to begin validating the FSD. It is hoped that efforts at INRIA will soon remedy this situation. In the meantime we are employing the denotational definition of TINY (provided in Appendix II of this report) to exercise our tools.

We have succeeded in processing the TINY definition through the parser, type-checker and compiler. The compiled definition has been run on the interpreter with several examples meant to test the various paths in the definition. The biggest TINY program we have executed under the AFDL interpretation of the definition has been one that accepts integers in the input stream and produces the corresponding factorials in the output stream. Since the TINY language does not have a multiplication operation, this was implemented via iterative addition! While such a program is quite small, the work done by the interpreter in terms of semantic function applications and the corresponding manipulation of static and dynamic chains is not trivial. Thus it has proven to be a fairly good test of the software.

Since AFDL + contains the necessary idioms for writing denotational semantic definitions, we expect that our software could also be useful to the scientific community at large.
2. A DESCRIPTION OF AFDL +: THE ADA FSD METALANGUAGE

2.1 OVERVIEW

The purpose of this chapter is to describe the semantic metalanguage in which the formal definition of Ada [1, 10] is written. This language, called AFDL (Ada Formal Definition Language) is essentially a typed lambda calculus with Ada syntactic sugaring.

The original version of AFDL was conceived by the INRIA team that wrote the Ada formal definition [10]. Certain enhancements of AFDL were found to be desirable by the ISI group whose goal is to build tools to test the Ada FSD; this enhanced version of AFDL is called "AFDL +". AFDL is a proper subset of AFDL +, and therefore AFDL programs are completely "upward compatible" with AFDL +. It is this enhanced version of AFDL that will be described in this chapter.

AFDL + and AFDL are purely applicative languages intended for writing denotational semantic definitions in an Ada-like syntax. AFDL contains functions, blocks, conditional and case "statements" (which are actually expressions), simple expressions, and packages (for modularity and information hiding). AFDL's basic data types are the integers and booleans, and its data type constructors are enumeration of types and (unlike Ada) function types. Conspicuously absent from AFDL are sum (union), product (record), and sequence (array) types; these types are basic to the denotational semantics method. As a result, the data types upon which the Ada FSD is based are only defined informally in plain language, and are not formally defined in AFDL (or in any other language). This situation is an obstacle to full understanding of the Ada FSD; moreover, it must be remedied before the FSD can be tested. AFDL + is an upward compatible extension of AFDL designed to overcome these shortcomings by including sum, product, and sequence types together with their associated operations. This allows the entire FSD to be formally defined, by writing it in AFDL +. This will facilitate human understanding of the FSD, and will also render it machine executable.

2.1.1 Style of this Description

The description of AFDL + presented in this chapter deals with two principal aspects of AFDL +: its syntax and semantics.

The syntax of AFDL + is given in two forms: (1) an LR(1) concrete syntax, which is used to parse AFDL + programs and translate them into equivalent abstract syntactic form, and (2) an abstract syntax, which is the target representation of this translation, and which is used as an intermediate form for type checking and compiling AFDL + programs into equivalent AFDL machine programs for execution of semantic descriptions written in AFDL +. The correspondence between concrete and abstract syntax is in most cases quite straightforward and requires no additional explanation. Those occasional cases that are not so clear are briefly explained in accompanying comments.

A summary of the AFDL + abstract syntax is given in Appendix III. A precise specification of the correspondence between AFDL + concrete and abstract syntax is given in Appendix IV. The correspondence consists of a specification input to the grammar analysis program LR; this kind of specification is explained in the documentation of the AFDL/Ada syntax tools [4]. Appendix IV presents the output from LR.
The semantics of AFDL + is given in plain English. A more precise, though complex, operational semantics of AFDL + is defined by the combination of the AFDL + parser, compiler, and the LISP code that implements the AFDL virtual machine. These processors are documented in [4, 5, 6], respectively.

2.1.2 Abstract Syntax Notation

The notation used to describe AFDL + abstract syntax is similar to BNF, except "::=" is used instead of ":: =". Lower case names denote syntactic categories or classes, and upper case names denote syntactic constants that are used as node names or labels. The suffix "-s" denotes a list of syntactic objects of the class to which the suffix is appended (e.g., id_s denotes a list of objects of class id). Most comments appear at the end of lines, and are prefixed by "--".

Abstract syntax constructs are basically trees implemented in a simple list structure of the form

(NAME son1 son2 ... sonN)

where NAME is the root node label and son1, ..., sonN are trees that are the immediate successors of the root node. When N = 0, the tree is simply represented as NAME rather than (NAME).

Several syntax classes are terminal, in that they represent sets of syntactic constants in AFDL +. These are

- var_id variable identifier; corresponds to ".varname" in concrete syntax.
- fun_id function or package identifier; corresponds to ".procname" in concrete syntax.
- type_id type identifier; corresponds to ".typemark" in concrete syntax.
- const_id (enumerated type) constant name; corresponds to ".etconst" in concrete syntax.
- integer integer; corresponds to ".integer" in concrete syntax.
- string string; corresponds to ".string" in concrete syntax.

2.2 LEXICAL ELEMENTS

AFDL has a simple set of lexical elements[4]. In addition to the usual delimiters such as parentheses, operators, etc., AFDL has four major disjoint classes of lexical elements:

- keywords
- names
- (positive) integer constants
- strings
Basically, keywords and names are identifiers of various kinds. In order to define their representations precisely, the following definitions are useful. A word is a nonempty sequence of letters; an upper (lower) case word is a word in which all the letters are upper (lower) case. An identifier is a nonempty sequence of letters, decimal digits, and underscore characters that must begin with a letter and not end with an underscore.

An AFDL keyword is a lower case word prefixed by an underscore; this representation of keywords causes them to be printed as boldface in the Ada FSD.

An AFDL integer constant is a nonempty sequence of decimal digits.

AFDL names are partitioned into four classes:

- names of packages and main functions
- names of values and auxiliary functions
- names of syntactic constructs and selectors
- type names

AFDL package names and function names are upper case identifiers; these constitute the lexical class ".procname" in the concrete syntax. AFDL value names and auxiliary function names are lower case identifiers; these constitute the lexical class ".varname" in the concrete syntax. AFDL syntactic construct names and selector names are lower case identifiers prefixed with a tilde (~); these constitute the lexical class ".etconst" in the concrete syntax. Syntactic construct and selector names print as lower case italic in the Ada FSD. An AFDL type name is an upper case identifier prefixed by a tilde (~); these constitute the lexical class ".typemark" in the concrete syntax. In order that type names be easily distinguishable, the leading tilde of a type name is changed to ".#" by the AFDL parser (during formation of AFDL abstract syntax trees), and therefore in the AFDL abstract syntax, type names are upper case identifiers prefixed by a ".#".

### 2.3 DECLARATIONS AND TYPES

A declaration is an entity used to associate a name with an object that it represents. Most AFDL declarations have two parts: a required specification part and an optional initialization or body part. In addition, AFDL has several kinds of type declarations; these are necessary to support the denotational semantics application for which AFDL is used.

#### 2.3.1 Declarations

In AFDL, four kinds of items are declared: variables ("objects" in Ada), functions (Ada value-returning subprograms), packages (of data types together with associated functions) to support the formal definition, and types.

#### 2.3.1.1 Variable declarations

AFDL has a standard kind of variable declaration, consisting of the variable's name, type, and an optional initial value to be assigned to that variable. A variable's type may be regarded as its
"signature". Variables declared within functions are generally initialized, whereas those declared within packages are not. Variable declarations without initializations are called variable specifications.

**CONCRETE SYNTAX**

variable_specification :: = .varname : type_id
                         | 2_var_id_list : type_id

variable_declaration :: = .varname : type_id := expression

2_var_id_list :: = .varname , .varname
                  | 2_var_id_list , .varname

**ABSTRACT SYNTAX**

var_spec: (VSPEC var_id_s type_id)

var_decl: (VDECL var_id type_id expr)

2.3.1.2 Function declarations

AFDL function declarations consist of at least a function specification, which consists of the function's name and result type together with the names and types of its formal parameters, plus an optional "initialization", the function's body. Function specifications are used to provide their names and "signature" (i.e., their argument and result types) in packages and also in situations where specifications are necessary to satisfy the "prior declaration" requirement of Ada when functions are declared to be mutually recursive.

**CONCRETE SYNTAX**

function_specification :: = function id formal_part return_type

function_declaration :: = function_specification is function_body

formal_part :: =               -- empty
                | ( parameter_declaration_list )

parameter_declaration_list :: = parameter_declaration
                              | parameter_declaration_list , parameter_declaration

parameter_declaration :: = name_list : type_id

return_type :: = return type_id

function_body :: = declaration_list block_body
                  | block_body

declaration_list :: = declaration ;
                    | declaration_list declaration ;
14 TOOLS FOR TESTING DENOTATIONAL SEMANTIC DECLARATION

ABSTRACT SYNTAX

fun_spec: (FSPEC fun_id fun_type)
fun_decl: (FDECL fun_id fun_type fun_def)
fun_type: (MAP type_id_s type_id)
fun_def: (LAM ids expr)

In the above abstract syntax, the fun_type is a function's "signature", i.e., the types of its formal parameters and the type of its result. In addition, a function declaration (in contrast to a function specification) has a fun_def, which is similar to a lambda calculus abstraction, whose bound variables are the function's formal parameters and whose body is the function's body.

Like Ada, but unlike lambda calculus, AFDL functions have their actual parameters transmitted by value. This causes difficulties when AFDL is used to express certain common denotational semantics idioms that assume evaluation (or parameter transmission) by name. As a result, "programming" denotational semantics in AFDL must be done with due recognition of this fundamental difference between AFDL and lambda calculus.

2.3.1.3 Package declarations

Packages are used in AFDL to encapsulate data types (domains and associated functions) that serve as "utility support" for the FSD. There are packages that define the Ada abstract syntax (Appendix G of the Ada FSD [10]), support for the Ada static semantics (Appendix H), support for the dynamic semantics (Appendix I), and basic denotational semantics notions such as environments, a store, and continuations (Appendices J and K). Packages are made visible to one another by means of use clauses. As in Ada, package declarations can have only a specification part (called a package specification) and an optional body part (a package body).

Packages are only syntactic sugar. The package structure cannot be used to resolve ambiguity. All package declarations should be viewed as being global.

CONCRETE SYNTAX

package_specification ::= package .procname is declarative_item_list end function_name_option

package_body ::= package body .procname is declarative_part end function_name_option

declarative_item_list ::= declarative_item ;
| declarative_item_list declarative_item ;

declarative_part ::= declarative_item ;
| package_body ;
| declarative_part declarative_item ;
| declarative_part package_body ;
declarative_item ::= variable_specification
    | variable_declaration
    | function_specification
    | function_declaration
    | type_declaration
    | function_type_declaration
    | sum_type_declaration
    | product_type_declaration
    | sequence_type_declaration
    | package_specification
    | use_clause

type_declaration ::= type type_id is type_definition

use_clause ::= use pkg_id_list

**ABSTRACT SYNTAX**

pkg_spec: (PSPEC pkg_id decl_item_s)

pkg_body: (PBODY pkg_id decl_item_s)

dcl_item: var_spec
    | var_decl
    | fun_spec
    | fun_decl
    | type_decl
    | fun_type_decl
    | prod_type_decl
    | sum_type_decl
    | seq_type_decl
    | pkg_spec
    | pkg_body
    | use_clause

type_decl: (TDECL type_id type_def)

use_clause: (USE pkg_id_s)

2.3.1.4 Type declarations

AFDL+ has several kinds of type declarations: those for enumerated, private, function, sum, product, and sequence types. In addition, AFDL+ provides a type equivalence declaration to permit certain kinds of type conversions to be expressed.

**Enumerated Types**

An enumerated type definition is a list of constants. This kind of type is used in the Ada FSD primarily to denote sets, such as the set of node labels in Ada abstract syntax trees, where these...
labels control the selection of alternatives in an AFDL case statement contained in a semantic function of the Ada FSD.

CONCRETE SYNTAX

type_definition ::= enumerated_type_definition

enumerated_type_definition ::= ( const_id_list )
                           | ( var_id_list )

ABSTRACT SYNTAX

type_def: enum_type

e num_type: const_id_s
           | var_id_s

Private Type

The private type is present in AFDL packages to indicate those data types whose detailed structure has been left unspecified, in order "not to portray superfluous or extraneous details".

CONCRETE SYNTAX

type_definition ::= private

ABSTRACT SYNTAX

type_def: PRIVATE

Function Types

Function types are present in AFDL, but not in Ada. Unlike Ada, AFDL can have function-valued parameters and return function-valued results, and function types are necessary to indicate this. In particular, denotational semantics notions that are intrinsically function-typed, such as continuations, are declared to have function types in AFDL.

CONCRETE SYNTAX

function_type_declaration ::= function type type_id (parameter_declaration_list ) return_type

ABSTRACT SYNTAX

fun_type_decl: (FNTDECL type_id fun_type)
            fun_type: (MAP type_id_s type_id)
A function type declaration specifies the signature of a functional type; its abstract syntactic structure is similar to that of a function specification.

**Sum, Product, and Sequence Types**

These types, present in AFDL+ but not in AFDL, were added to permit the complete specification of the Ada FSD to be done in AFDL+ itself. The inclusion of sum, product, and sequence types in AFDL+ make it possible to express in the semantic metalanguage essential details currently concealed in private types. A sum (product) type is declared as the sum (product) of a finite number of other types; a sequence type is declared as a finite sequence of another type. In addition to these type-formation operations, other operations are provided. For sum types, projection, injection, and domain query operations are provided. For product types, selection and tupling operations are provided. For sequence types, "head", "tail", concatenation, and sequence construction operations are provided.

**CONCRETE SYNTAX**

```
sum_type_declaration ::= sum type type_id (2_type_id_list)
product_type_declaration ::= product type type_id (2_type_id_list)
sequence_type_declaration ::= sequence type type_id type_id

2_type_id_list ::= type_id , type_id |
| 2_type_id_list , type_id
```

**ABSTRACT SYNTAX**

```
sum_type_decl: (SMTDECL type_id type_id_s)
prod_type_decl: (PRTDECL type_id type_id_s)
seq_type_decl: (SQTDECL type_id type_id)
```

Sum and product types $T$ with component types $T_1, \ldots, T_n$, where $n \geq 2$, are declared via

- **sum type** $T (T_1, \ldots, T_n)$
- **product type** $T (T_1, \ldots, T_n)$

where the type identifiers $T_i$ are all distinct.

A sequence type $T$ with components of type $S$ is declared via

- **sequence type** $T S$

**Type Equivalence Declarations**

A type $T$ may be declared *equivalent* to a type $S$ by the declaration

- **type** $T$ is $S$.

See Section 2.4.3 for a discussion of the application of type equivalence declarations.
CONCRETE SYNTAX

type_definition ::= type_id

ABSTRACT SYNTAX

type_def: type_id

2.3.1.5 Order of declarations

At the global level, the order of declarations is irrelevant, except that specifications for variables or functions may not occur after the definitions of those objects. Thus "forward declarations" are legal, but unnecessary at this level.

At the local level (i.e., inside functions and variables), the order is relevant. Names are only visible once they are declared. The name of a variable is NOT visible within its own declaration, but the name of a function is. Forward declarations for functions (i.e., occurrences of function specifications (FSPECs)) are legal and necessary to effect mutual recursion at a local level. Forward declarations for variables (VSPECs) are not legal. A name may not be declared (with the same type) twice within the same block.

At the global level, variables must not be mutually recursive, i.e., there must be some partial ordering on which variables require the values of other variables in order to produce their own value. This is allowed to be value dependent -- it is as liberal as possible, and is not (and could not be) checked statically. (At present, infinite recursion would result if the value of a variable that was recursively defined in terms of other variables were requested (unless shielded properly with function bodies).)

At the local level, functions must be defined (FDECLed) before they are "used". This is only a problem when variable declarations and function specifications/declarations are intermixed. The actual restriction here is conservative in that it outlaws some programs that could be legal at runtime. A function is considered "used" at the point when a reference to it (not necessarily a call) is contained in the body of a variable declaration (even if shielded within a function declaration that occurs within that variable declaration), or if it occurs within another function that itself has been "used" by this definition. Note the recursion in this definition. If the initialization for a variable x contains a reference to a function f, and f contains a reference to g, and g contains a reference to h, and h is not defined before the declaration of x, then the program is illegal. These rules allow mutually recursive functions to be declared. Note the distinction between visibility of function names (forward declarations make a function name visible) and the definition of the corresponding functions (only FDECLs do that).

The general idea is that order is not required at the global level, and it is not checked statically that variables do not depend, in a mutually recursive manner, on one another; where at the local level, proper order is required, and some (necessarily conservative) static checking is performed. These choices are largely pragmatic: we found that the global level of the Ada FSD was NOT properly ordered, and performing static checking appeared to be expensive. The checking at the local level catches some errors and allows the linear elaboration of declarations in the compiled code.
2.4 EXPRESSIONS

AFDL+ is purely applicative, and thus expressions are the principal computational mechanism in the language. AFDL has two kinds of expressions: "simple" expressions, which are an extended subset of Ada expressions, and "compound" expressions, most of which correspond to certain Ada compound statements. Thus blocks, case "statements", conditional "statements", etc., all yield values. Requiring the return expression to be used to return a value from a function as in AFDL is excessively verbose, and thus return expressions are treated as comments in AFDL+. Finally, semicolons are made optional in AFDL+ where they were formerly required as statement terminators in blocks, case expressions, and conditional expressions in AFDL. These changes are "upward compatible" in the sense that syntactically legal AFDL programs are still syntactically legal in AFDL+.

**CONCRETE SYNTAX**

expression ::= return_expr

| relation
| and_expression
| or_expression
| xor_expression
| andthen_expression
| orelse_expression

and_expression ::= relation and relation

| and_expression and relation

or_expression ::= relation or relation

| or_expression or relation

xor_expression ::= relation xor relation

| xor_expression xor relation

andthen_expression ::= relation and then relation

| andthen_expression and then relation

orelse_expression ::= relation or else relation

| orelse_expression or else relation

relation ::= simple_expression

| simple_expression relop simple_expression
| simple_expression elt type_id

simple_expression ::= sum

| simple_expression seq_op sum
| simple_expression dom_op type_id
\[ \text{sum} ::= \text{term} \\
| \text{un_op term} \\
| \text{sum add_op term} \]

\[ \text{term} ::= \text{primary} \\
| \text{term} \ast \text{primary} \]

\[ \text{primary} ::= \text{id} \\
| \text{integer} \\
| \text{string} \\
| (\text{expression}) \\
| ( [ \text{expression}_\text{list} ] ) \\
| ( [ \text{expression}_\text{list} ] ) \\
| (\text{type_coercion}) \\
| \text{if_expr} \\
| \text{case_expr} \\
| \text{type_case_expr} \\
| \text{block} \\
| \text{function_call} \]

**ABSTRACT SYNTAX**

\[ \text{expr} : \text{simple_expr} \]

\[ \| (\text{CAST type_id expr}) \quad \text{-- type coercion} \]

\[ \| (\text{IF expr expr expr}) \quad \text{-- conditional} \]

\[ \| (\text{CASE expr alt_s}) \quad \text{-- case} \]

\[ \| (\text{TCASE expr t_alt_s}) \quad \text{-- type case} \]

\[ \| (\text{LET decl_s expr}) \quad \text{-- block} \]

\[ \| (\text{APPLY expr expr_s}) \quad \text{-- function call} \]

\[ \| (\text{WARNING expr msg}) \quad \text{-- from static type checker} \]

\[ \text{simple_expr} : \text{(binop expr expr)} \]

\[ \| (\text{unop}) \]

\[ \| (\text{ELT expr type_id}) \]

\[ \| (\text{INJ expr type_id}) \]

\[ \| (\text{INJ expr type_id} (\text{FROM type_id})) \]

\[ \| (\text{PRO expr type_id}) \]

\[ \| \text{id} \]

\[ \| \text{integer} \]

\[ \| \text{string} \]

\[ \| (\text{PRDEN expr_s}) \]

\[ \| (\text{SQDEN expr_s}) \]

Return, type coercion, conditional, case, type case, block, and function call expressions are considered to be "compound" expressions; their concrete syntax is given in Section 2.4.4.
2.4.1 Names

**CONCRETE SYNTAX**

name ::= .varname
   | .procname

id ::= name
   | const.id

function_name_option ::= -- empty
   | id

const_id ::= .etconst

type_id ::= .typemark

**ABSTRACT SYNTAX**

id: var_id
   | fun_id
   | const_id
   | type_id

2.4.2 Operators

The equality comparison of "infinite" values, i.e., values that include closure objects, is illegal.\(^1\) All other values may be compared for equality (this currently is a run-time check in the AFDL virtual machine, although it could be done statically).

**CONCRETE SYNTAX**

relop ::= =
   | /=
   | >=
   | <=

un_op ::= -
   | not
   | length

add_op ::= +
   | -

\(^1\)Closure objects are representations of function objects. A closure object is a pair of pointers; the first pointer points to a corpus to be executed within an environment pointed to by the second pointer (see [6]). Closure objects can result in circular list structures in their Lisp representations.
2.4.3 Simple Expressions

Simple expressions in AFDL+ are used to indicate the application of basic operations to appropriate operands. The standard basic operations in AFDL are

integer arithmetic  binary addition, subtraction, and multiplication; unary negation

boolean operations  conjunction, disjunction, exclusive or, complement, andthen, orelse

integer relations  equal (=), not equal (/=), greater than (>), greater than or equal (>=), less than (<), less than or equal (<=)

Additional domain operations added in AFDL+ for sum, product, and sequence types are

sum types  domain injection (inj), projection (pro), membership test (elt)

product types  domain element selection, tupling

sequence types  "head" (:), "tail" (::), concatenation (&), length (length), sequence formation

AFDL+ provides several sum domain operations. If $T$ is declared via

**sum type** $T$ ($T_1, \ldots, T_n$),

then $x$ inj $T$ injects an element of component type $T_i$ into the sum type $T$; $x$ elt $T_i$ tests whether an element $x$ of sum type $T$ was injected from component type $T_i$; $x$ pro $T_i$ projects an element $x$ of sum type $T$ into component type $T_i$ if $x$ elt $T_i$ is true, and otherwise causes a fatal error in the AFDL+ virtual machine.
If the product type $T$ is declared via

\texttt{product type } T (T_1, \ldots, T_n),

then an element of $T$ is constructed from elements $x_i$ of type $T_i$ by the tupling operation $(x_1, \ldots, x_n)$. The $i$-th component of a value $x$ of product type $T$ is selected by the operation $x.i$, where $i$ must be a constant (literal).

A sequence type $T$ with components of type $S$ is declared via

\texttt{sequence type } T S.

An element of sequence type $T$ is constructed from elements $x_i$ of type $S$ by the operation $[x_1, \ldots, x_n]$, $n \geq 0$. $[]$ denotes the empty element of any sequence type. The $i$-th component (head) of an element $x$ of sequence type $T$ is selected by the operation $x.i$, where $i$ may be any integer expression. The $i$-th tail of $x = [x_1, \ldots, x_n]$ is selected by the operation $x::i$, and is equal to $[x_{i+1}, \ldots, x_n]$ (if $i = n$ then $x::i$ is equal to $[]$). If $i$ is "out of range" ($\leq 0$ or $> n$) for $: : i$, then a fatal error occurs in the AFDL virtual machine.

A type $T$ may be declared \textit{equivalent} to a type $S$ by the declaration

\texttt{type } T \texttt{is } S.

This permits an element $x$ of type $T$ to be converted to type $S$ by applying the \texttt{cast} $S(x)$; conversely, an element $y$ of type $S$ can be converted to type $T$ via the \texttt{cast} $T(y)$. Type conversions must be explicit and have no run-time significance. Equivalent types provide a way of declaring a sum type $T$ with multiple components $T_i$ that are equivalent to some type $S$. If $x \in T$, then $S(x \text{ pro } T_i)$ is type $S$, and if $y$ is an element of type $S$, then $(T.(y)) \text{ inj } T$ is an element of type $T$.

2.4.4 Compound Expressions

Compound expressions in AFDL provide return expressions, conditional (\texttt{if-then-else}) expressions, "case" expressions, blocks and local declaration of variables (similar to the "let variable = expression in expression" capability in lambda calculus), and function application. An additional compound expression in AFDL is a "type case" expression, in which the case selection expression must evaluate to a value in a sum domain, and a case alternative is selected on the basis of which particular constituent type the case expression possesses.

CONCRETE SYNTAX

\texttt{return expr :: = return expression}

\texttt{if expr :: = if expression then expression semi if_expr_tail}

\texttt{if_expr_tail :: = else expression semi end if}

\texttt{| elsif expression then expression semi if_expr_tail}

\texttt{case_expr :: = case expression is alternative_list end case}

\texttt{alternative_list :: = alternative}

\texttt{| alternative_list alternative}
alternative ::= when choice_list => expression semi

choice_list ::= choice
   | choice_list | choice

choice ::= const_id
   | .varname
   | others

type_case_expr ::= tcase expression is type_alt_list end tcase

type_alt_list ::= type_alt
   | type_alt_list type_alt

type_alt ::= when .varname : type_id => expression semi
   | when_others .varname : type_id => expression semi

when_others ::= when others

block ::= declare declaration_list block_body
   | block_body

block_body ::= begin expression semi end function_name_option

type_coercion ::= type_id ( expression )

semi ::= -- empty
   | ;

ABSTRACT SYNTAX

expr: (IF expr expr expr) -- conditional
   | (CASE expr alt_s) -- case
   | (TCASE expr t_alt_s) -- type case
   | (LET decl_s expr) -- block
   | (CAST type_id expr) -- type coercion
   | (WARNING expr msg) -- constructed by static type checker

alt: (choice_s expr)

choice: id
   | OTHERS

t_alt: (ALT var_id type_id expr)
   | (OTHERS var_id type_id expr)

The AFDL > type case expression is a variant of the usual case construct and is provided to simplify the manipulation of sum types. The tcase (type case) expression
DEFINITIONS OF PROGRAMMING LANGUAGES

tcase expr is
   when t_1 : T_1 => expr_1 ;
   ...
   when t_n : T_n => expr_n ;
   when others t : T => default
end tcase

evaluates expr, in a new scope in which t_i is bound to expr pro T_i, if expr elt T_i, and yields the resulting value as the value of the type case expression. If a when clause for the appropriate T is not provided, then if the others clause is present, "default" is evaluated in a scope in which t is bound to expr, and the resulting value is yielded as the value of the type case expression; otherwise, a fatal error occurs.

2.5 FUNCTIONS

Functions in AFDL + are like those in Ada, except that AFDL + functions can take function-valued parameters and return function-valued results. This enhancement enables AFDL + to be used for programming denotational semantic definitions. Actual parameters of AFDL + functions are bound to corresponding formal parameters by value; consequently, programming denotational semantic definitions in AFDL + must be done carefully, in order to avoid using lambda calculus idioms whose correctness depends upon parameter binding by name.

A common situation in which transmission of actual parameters by value would lead to nontermination whereas transmission by name would not, is the evaluation of the semantics of a recursively defined construct such as a while loop, using continuation semantics. The evaluation of the semantics of a while loop requires evaluation of the value of the test expression followed by the semantic evaluation of the loop body if the test value is true. This latter semantic evaluation invokes a function that takes as an actual parameter the continuation (a function) that continues the computation after the traversal of the body. This continuation (recursively) computes the semantics of the entire while loop (given a new state). If an attempt is made to first evaluate this continuation (which represents the entire possible future behavior of the while loop), then an infinite semantic computation will result. The usual way to avoid this phenomenon is to simulate parameter transmission by name by "encapsulating" the offending actual parameter inside a function literal (sometimes called a "thunk"), producing a closure object as an actual parameter value and thereby "suspending" further evaluation involving this parameter until it is later applied to a state.

A detailed illustration of this situation, in the context of the formal semantic definition of Gordon's pedagogical programming language TINY [9], is given in [2].

CONCRETE SYNTAX

function_call ::= = id ( )
   | id ( expression_list )
   | curried_function_call ( )
   | curried_function_call ( expression_list )
curried_function_call ::= id ( )
  | id ( expression_list )
  | curried_function_call ( )
  | curried_function_call ( expression_list )

ABSTRACT SYNTAX

expr: (APPLY expr expr_s)

AFDL+ functions have their actual parameters bound by value, as previously mentioned. To facilitate the programming of denotational semantics, AFDL+ functions are curried.

2.6 PACKAGES

AFDL+ packages, like those in Ada, are used to specify collections of logically related entities. In the Ada FSD, packages are used to define abstract data types that support the functions that comprise the main portion of the FSD. In particular, the Ada FSD contains packages that define the Ada abstract syntax, the static and dynamic environments, and continuations and an abstract store.

The syntax of packages is described in Section 2.3.1.3.

2.7 PROGRAM STRUCTURE

AFDL+ program structure reflects that of the Ada FSD. The main corpus of the FSD consists of a nonempty sequence of (mutually recursive) function declarations. The "support" for the FSD, contained in appendices, consists of nonempty sequences of package declarations and package bodies. An AFDL+ "compilation unit", therefore, is a function declaration, a package declaration, or a package body; an AFDL+ program is a nonempty sequence of compilation units.

CONCRETE SYNTAX

afdl_program ::= compilation_unit_list

compilation_unit_list ::= compilation_unit ;
  | compilation_unit_list compilation_unit ;

compilation_unit ::= function_declaration
  | package_specification
  | package_body

ABSTRACT SYNTAX

program: comp_unit_s

comp_unit: fun_decl
  | fun_spec
  | pkg_spec
  | pkg_body
3. AFDL + TRANSCRIPTS

This chapter consists of several AFDL + [3] transcripts in which a definition of Gordon’s example programming language “TINY” ([9], pp. 57-61) is “debugged.” Three versions of the Tiny definition are used for illustration as follows:

- Version 1 of the Tiny definition contains a type error that is detected statically by the AFDL + Typechecker.
- Version 2 of the definition, in which this type error has been corrected, contains no type errors, but results in infinite recursion due to AFDL + ’s call-by-value parameter passing rule.
- Version 3, the final version of the definition, in which an expression that yields a functional value has been “shielded” by a function definition, executes correctly on a sample Tiny program.

We have made efforts to include enough comments in the transcripts so that this chapter stands on its own. However, if more than a superficial understanding of the contents is desired, we expect the reader to be familiar with the Interlisp language, the AFDL + Typechecker [7], Compiler [5], and Virtual-Machine [6] documentation.

We first present the third and final version of the AFDL + definition of Tiny. This is the version without errors. The differences between the correct Version 3 and the incorrect Versions 1 and 2 will be discussed subsequently.

```
package TINY is
  -- Syntactic Domains

  sum type EXPR (INTEGER, BOOLEAN, IDENT, READ, NOT, EQUAL, PLUS);

  type IDENT is STRING;
  type READ is private;
  type NOT is EXPR;
  product type EQUAL (EXPR, EXPR);
  product type PLUS (EXPR, EXPR);

  sum type COM (ASSIGN, OUTPUT, IF, WHILE, SEQ);

  product type ASSIGN (IDENT, EXPR);
  type OUTPUT is EXPR;
  product type IF (EXPR, COM, COM);
  product type WHILE (EXPR, COM);
  product type SEQ (COM, COM);

  -- Semantic Domains

  product type STATE (MEMORY, INPUT);
```
function type MEMORY (id: IDENT) return VALUE_OR_UNBOUND;
sum type VALUE_OR_UNBOUND (VALUE, UNBOUNDVALUE);
type UNBOUND_VALUE is (unbound);

sequence type INPUT VALUE;

sum type VALUE (INTEGER, BOOLEAN);

function type CONT (state: STATE) return ANS;
function type ECONT (value: VALUE) return CONT;

sum type ANS (FINAL_ANSWER, PARTIAL_ANSWER);
type FINAL_ANSWER is (error, stop):
product type PARTIAL_ANSWER (VALUE, ANS);

-- Auxiliary Functions

function update (memory: MEMORY; value: VALUE; id: IDENT)
  return MEMORY is
  function new_memory (ident: IDENT) return VALUE_OR_UNBOUND is
    begin
    if ident = id
      then value inj VALUE_OR_UNBOUND
      else memory(ident)
    end if
    end new_memory;
  begin
    new_memory
  end update;

function value_of (memory: MEMORY; id: IDENT) return VALUE_OR_UNBOUND is
begin
memory(id)
end value_of;

function error (state: STATE) return ANS is
begin
  error inj ANS
end error;

function final (state: STATE) return ANS is
begin
  stop inj ANS
end final;

function initial_memory (id: IDENT) return VALUE_OR_UNBOUND is
begin
  unbound inj VALUE_OR_UNBOUND
end initial_memory;
null
when equal: \textit{EQUAL} =>
\begin{align*}
\text{EVALEXPR} \\
\text{( equal:1,} \\
\text{declare} \\
\text{function econt1 (value1: VALUE) return CONT is} \\
\text{begin} \\
\text{EVALEXPR} \\
\text{( equal:2,} \\
\text{declare} \\
\text{function econt2 (value2: VALUE) return CONT is} \\
\text{begin} \\
\text{econt1 (value1 = value2) inj VALUE} \\
\text{end econt2;} \\
\text{begin} \\
\text{econt2} \\
\text{end}) \\
\text{end econt1;} \\
\text{begin} \\
econt1 \\
\text{end)};
\end{align*}
\begin{align*}
\text{when plus: } \textit{PLUS} => \\
\text{EVALEXPR} \\
\text{( plus:1,} \\
\text{declare} \\
\text{function econt1 (value1: VALUE) return CONT is} \\
\text{begin} \\
\text{EVALEXPR} \\
\text{( plus:2,} \\
\text{declare} \\
\text{function econt2 (value2: VALUE) return CONT is} \\
\text{begin} \\
\text{if value1 elt INTEGER and} \\
\text{value2 elt INTEGER} \\
\text{then econt((value1 pro INTEGER) +} \\
\text{(value2 pro INTEGER))} \\
\text{inj VALUE) } \\
\text{else error} \\
\text{end if} \\
\text{end econt2;} \\
\text{begin} \\
\text{econt2} \\
\text{end}) \\
\text{end econt1;} \\
\text{begin} \\
econt1 \\
\text{end)} \\
\text{end tcase} \\
\text{end EVALEXPR;}
\end{align*}
function EVAL_COM (com: COM; cont: CONT) return CONT is
begin
tcase com is
    when ass: ASSIGN =>
        EVAL_EXPR
        ( ass:2,
         declare
         function econt (value: VALUE) return CONT is
            function new_cont (state: STATE) return ANS is
                begin
                    cont( (update(state:1, value, ass:1), state:2) )
                end new_cont;
                begin
                    new_cont
                end econt;
                begin
                    econt
                end);
    when out: OUTPUT =>
        EVAL_EXPR
        ( EXPR(out),
         declare
         function econt (value: VALUE) return CONT is
            begin
                function new_cont (state: STATE)
                    return ANS is
                        begin
                            (value, cont(state)) inj ANS
                        end new_cont;
                        begin
                            new_cont
                        end econt;
                        begin
                            econt
                        end);
    when if: IF =>
        EVAL_EXPR
        ( if:1,
         declare
         function econt (value: VALUE) return CONT is
            begin
                if value elt BOOLEAN
                then
                    if value pro BOOLEAN
                    then EVAL_COM(if:2, cont)
                    else EVAL_COM(if:3, cont)
                end if
            end if
else error
end if
end econt;
begin
econt
end;
when while: WHILE =>
declare
EVAL_EXPR
( while:1,
declare
function econt (value: VALUE) return CONT is
begin
if value elt BOOLEAN
then
if value pro BOOLEAN
then EVAL_COM
( while:2,
declare
function while_cont
( state: STATE) return ANS is
begin
EVAL_COM(com, cont)(state)
end while_cont;
begin
while_cont
end
)
else cont
end if
else error
end if
end econt;
begin
econt
end
)
when seq: SEQ =>
EVAL_COM(seq:1, EVAL_COM(seq:2, cont))
extcase
end EVAL_COM;
end TINY;

Version 1 of the definition differs from Version 3, shown above, in two respects. First, the "IDENT" alternative of the "tcase" of "EVAL_EXPR" consists of the following, in which the curried application of a continuation to "state" in the 8th line has been omitted (i.e., commented out). This results in a type mismatch since a value of type "CONT" is yielded where "ANS" is required.

when ident: IDENT =>
Second, the "WHILE" alternative of the "tcase" in "EVAL_COM" consists of the following, in which the continuation-valued actual parameter "EVAL_COM(com, cont)" (passed by value to the invocation of "EVAL_COM" that evaluates the effect of traversing the body of a "while" loop) is not "shielded" by the function "while_cont". This will result in infinite recursion when the Tiny definition is applied to Tiny programs that contain "while" loops in which the boolean expression parts do not reference or modify the state (i.e., contain no variable references or "read" expressions). The introduction of "while_cont" simulates passing "EVAL_COM(com, cont)" by name.

when while: WHILE = >
  EVAL_EXPR
  ( while:1,
    declare
      function econt (value: VALUE) return CONT is
      begin
        if value elt BOOLEAN
          then if value pro BOOLEAN
                then EVAL_COM
                    (while:2,
                     EVAL_COM(com, cont)
                    )
                else cont
                end if
            else error
            end if
          end if
        end cont;
      begin
        econt
      end
    end;
while true
   do
      num := read; res := 1; i := 2;
      
      while not (i = (num + 1))
         do
            x := res; j := 1;
            
            while not (j = i)
               do
                  res := res + x; j := j + 1
               od;
            
            i := i + 1
         od;
   output res
od

In the following we present five transcripts of operations on the TINY definition and the above program in the TINY language. In Transcript I we present the parsing of the TINY definition. For illustrative purposes, two syntax errors were introduced into Version 3 of the TINY definition. These are detected during the parse. In Transcript II the static semantic error in Version 1 of the TINY definition is detected. In Transcript III Version 2 of the TINY definition passes the static semantic check and is compiled and run on the virtual-machine. During the run an error is detected in the definition. Transcript IV shows the interpretation of Version 3 the TINY definition after this "infinite recursion" error has been corrected. In this transcript the TINY definition is made to interpret the TINY program shown above. In the final transcript, Transcript V, we illustrate the capability of the virtual-machine to optimize tail recursion. This capability can be turned on or off by the user at run time by setting or resetting a flag and is expected to be useful in improving efficiency during interpretation of large semantic definitions. The correct version of the TINY definition, Version 3, is used for Transcript V.

NOTE: Comments appear in italics in all of the transcripts.

Transcript I: Parsing and Storage of Abstract Syntax Trees

In this transcript we show the process of parsing Version 3 of the TINY definition into which two minor syntax errors were deliberately introduced.

We first try to parse the text of the TINY definition without generating the abstract syntax trees. This is done by making the second parameter to the parse* command be NIL i.e., by omitting it. The parsing is faster as a result. The first error found is a semicolon where a colon should appear.

parse* TINY-DEFINITION.TXT
Parsing file <AU-ADA>TINY-DEFINITION.TXT.1

.................

NOTE: Comments appear in italics in all of the transcripts.
DEFINITIONS OF PROGRAMMING LANGUAGES

***ERROR*** LINE 47: Syntax error in state 39
help!

(HELP broken)
2:quit
NIL

The parse error puts us in a Lisp break. We exit Lisp, fix up the parse error just found, and return to Lisp. We exit the break by typing a t and then redo the parse* command. The next error found is a misspelled type identifier; all type identifiers in AFDL are uppercase identifiers prefixed with a tilde.

3:t
3_redo parse*
Parsing file <AU-ADA>TINY-DEFINITION.TXT.2

_function error (state: -STATE) _return _ANS _is

***ERROR*** LINE 63: Syntax error in state 112
help!

(HELP broken)
4:quit
NIL

Again exit, fix up the error and return. The next redo of the parse* command finds no errors.

5:t
5_redo parse*
Parsing file <AU-ADA>TINY-DEFINITION.TXT.3
End of file <AU-ADA>TINY-DEFINITION.TXT.3
Done.

We do the parse again, this time providing T as the second parameter to the parse* command. This causes the abstract syntax trees to be generated. As each program unit is parsed (in this case only one, the package TINY), its name is printed out by the semantic action appended to the topmost concrete syntax production of the AFDL* language. During parsing the Control-L character is defined as an interrupt character. Typing a tL prints out the index of the line in the file on which the lexer is currently working, along with the load average.

6_parse* TINY-DEFINITION.TXT T
Parsing file <AU-ADA>TINY-DEFINITION.TXT.3
PARSING AT LINE 98, LOAD 10.9
PARSING AT LINE 126, LOAD 9.9
PARSING AT LINE 137, LOAD 10.0
PARSING AT LINE 194, LOAD 10.2
TINY
End of file <AU-ADA>TINY-DEFINITION.TXT.3
Done.
Next, we use the `saveafdl` command to save the generated abstract syntax trees in a structured file. The property list (CHAPTER SMALL) is given to each unit being stored. This is used by the Typechecker in reporting the full name of contexts in which errors exist (see Transcript II). The CHAPTER property is particularly useful in processing the Ada FSD; it enables each semantic function to be located to within a Chapter of the FSD. The third parameter T ensures that a new version of the output file is created. If it had been omitted or NIL, the abstract syntax trees being saved would have been added to the contents of the latest version of the output file.

```
7_saveafdl* TINY-DEFINITION.AS (CHAPTER SMALL)
...T
<AU-ADA>TINY-DEFINITION.AS.1
```

Transcript II: Static Semantic Error
Tiny Definition, Version 1

The transcript of the application of the AFDL + tools to Version 1 of the Tiny definition appears next. Not shown are the stages in which Version 1 of the AFDL + Tiny definition and the Tiny program are parsed and output to structured files (see Transcript I above). (A parser for Tiny was generated in the same way that AFDL + and Ada parsers have been generated.) Since a type-error is uncovered, the Compiler and Virtual-machine are not run in this example.

```
1_(TCTypeCheck 'TINY-DEFINITION.AS 'TINY-DEFINITION-SYMTAB]
?TC: "[Typelf:]" "[in:]" SMALL.TINY.EVAL_EXPR.ident.cont Incompatible then and else expressions
?TC: "[3Fun:]" "[in:]" SMALL.TINY Invalid function declaration "[name:]
EVAL_EXPR "[type:]" (FUN (#EXPR #ECONT)
#CONT)
?TC: "[Pass3:]" Error during Pass 3
?TC: "[TypeCheck:]" Fatal error
NIL

Run the Typechecker on the structured file "TINY-DEFINITION.AS" produced by the parser, producing a symboltable "TINY-DEFINITION-SYMTAB". An error is detected: the "then" and "else" parts of a conditional, in "ident.cont". in function "EVAL_EXPR", in package "TINY", and in chapter "SMALL", yield incompatible types.

2_(NIL (symtab _ (SYMOpenTable 'TINY-DEFINITION-SYMTAB]
NIL

Open the symboltable produced by the Typechecker. The function "NIL" is used to prevent values yielded by expressions from being printed.

3_(NIL (evalexpr _ (SYMGetValue symtab NIL T 'EVALEXPR):1]
NIL

Extract the body of "EVAL_EXPR" from the symboltable.

4_(EV evalexpr]
edit

Edit the function body. Travel to the "IDENT" alternative.
Print the "IDENT" alternative. It can be seen that a conditional is annotated with the error message that was printed above, and that the "then" part of that conditional yields a value of type "ANS" whereas the "else" part yields "CONT". Since a value of type "ANS" is required, the "else" part must be in error.
8*OK
evalexpr
   Exit the editor.

5_(SYMCloseTable symtab]
T
   Close the symboltable.

Transcript III: Dynamic Semantic Error
Tiny Definition, Version 2

The transcript of the application of the AFDL + tools to the second version of the Tiny definition, in which the above error had been corrected, appears next. Again, the parsing stages are not shown.

In this transcript, as well as in Transcript IV (following), certain compiler-generated unique compound names appear. The AFDL compiler [5] generates these names to uniquely identify contexts and declared objects within AFDL programs. For example, these unique names are useful in specifying which of several continuations, declared with the same name "continue1" (as is done in the Ada FSD), is being designated in a discussion or program trace. These unique names consist of several sections separated by periods, where each section is either an AFDL identifier or a non-negative decimal integer, e.g., EVAL_COM.4.1.econt.1.while_cont. These names are formed in the following way. The scope that is the entire body of a top-level AFDL function is named with the name of that top-level function itself. This scope is the one in which (conceptually) only formal parameters of the function are declared. Each subsequent nesting of a block within a scope introduces an additional period and section in the unique name. Blocks at the same nesting level are named by numbering them in order of occurrence, beginning with 1. The compiler produces code that introduces a new block (and hence nesting level) for

1. each LET (AFDL declare-begin-end construct), and
2. each alternative of a TCASE
in the AFDL abstract syntax.

Thus the compound name EVAL_COM.4.1.econt.1.while_cont is the fully-qualified name of a unique object that is possibly one of many objects declared with the same name while_cont. The fully-qualified name of an object provides a way of unambiguously locating it within an AFDL program. Thus in the above name, while_cont is located by going into the body of the top-level function EVAL_COM, then proceeding to the fourth block within it and to the first block within that block. A function econt will be found to be declared at this point. The designated object while_cont is located (declared) in the first block within econt.

The compiler-generated fully-qualified names are particularly useful for locating the body (program) part of a closure object. Internally, a typical closure object is of the form

(INT#ClosureObject (EVAL_COM 142) (INT#Frame ...))

where the body part (EVAL_COM 142) points to a location within EVAL_COM via an offset 142. This particular example corresponds to a closure object formed from a continuation declared within the body of the function EVAL_COM. Since there may be several such continuations the offset is a poor indication of which particular continuation is being pointed to. Therefore, a more readable form of
this pointer is the corresponding fully-qualified name. Accordingly, closure objects are displayed together with the fully-qualified name that designates their body part. Thus the above closure object would be displayed as (see step 12 of Transcript III below)

"EVAL_COM.4.1.econt" : (INT#ClosureObject ( ... ) ( ... ))

Transcript III now follows.

1_(TCTypeCheck 'TINY-DEFINITION.AS 'TINY-DEFINITION-SYMTAB]
Type-check the structured file "TINY-DEFINITION.AS", producing the symbol table "TINY-DEFINITION-SYMTAB". No errors.

2_compile* TINY-DEFINITION-SYMTAB
update 21 (FUN (#MEMORY #VALUE #IDENT) #MEMORY)
value_of 7 (FUN (#MEMORY #IDENT) #VALUE_OR_UNBOUND)
error 6 (FUN (#STATE) #ANS)
final 6 (FUN (#STATE) #ANS)
initial_memory 6 (FUN (#IDENT) #VALUE_OR_UNBOUND)
EVAL_EXPR 242 (FUN (#EXPR #ECONT) #CONT)
EVAL_COM 200 (FUN (#COM #CONT) #CONT)

0 Warnings
0 Fatal Errors

TINY-DEFINITION-SYMTAB -- Compilation complete.
NIL

Compile the symbol table. The compile* Lisp macro calls the function COM#CompileSymbolTable and if given a second argument of T would produce a compiler listing file. The integers in the second column are the number of instructions compiled for the corresponding symbol.

3_(tinystf_ (STFOpenFile 'TINY-TEST-AS.TXT]
Loading directory of TINY-TEST-AS.TXT
Last updated 10-Jun-82 11:50:18
TINY-TEST-AS.TXT#STFDIR

Open the structured file that contains the abstract syntax form of the Tiny program.

4_(NILL (tinyprogram_ (STFGetObject tinystf 'TinyProgram):1:1]
NIL

Extract the Tiny program from the structured file. The ":1:1" CLISP selection on the value returned by STFGetObject reflects the structure of the abstract syntax of the Tiny language. The first :1 is necessary because STFGetObject returns a list of items that are stored in the structured file on the key given as an argument to the function. An item is of the form (object prop1 value1 prop2 value2 ...). Therefore, the following :1 selects the body of the Tiny program as pretty printed below.
5_(PP tinyprogram]

[#WHILE
   (][(#BOOLEAN true)
   (][#SEQ
      (][#SEQ (([#ASSIGN ("num" (#READ NIL)))
         (][#ASSIGN ("res" (#INTEGER 1))))
      (][#ASSIGN ("i" (#INTEGER 2))))
   (][#WHILE
      (][#NOT (][#EQUAL (][(#IDENT "i")
         (][#PLUS (][(#IDENT "num")
            (][#INTEGER 1]
      (][#SEQ
         (([#SEQ (][(#ASSIGN ("x" (][#IDENT "res"))
            (][#ASSIGN ("j" (][#INTEGER 1))))
         (][#WHILE (][#NOT (][#EQUAL (][(#IDENT "j")
            (][#IDENT "i"))]
            (][#SEQ (][(#ASSIGN ("res" (][#PLUS (][(#IDENT "res")
               (][#IDENT "x"))])
            (][#ASSIGN ("j" (][#PLUS (][(#IDENT "j")
               (][#IDENT "i"))])
      (][#ASSIGN ("i" (][#PLUS (][(#IDENT "i")
         (][#INTEGER 1]
      (][#OUTPUT (][(#IDENT "res")
         (][#INTEGER 1]
   (][#OUTPUT (][(#IDENT "res"]

Pretty-print the Tiny program. The program has AFDL+ type "COM". Note the sum-
type tags such as " # SEQ".

6_(_NIL (][#CreateMachineState (][#SYMOpenTable 'TINY-DEFINITION-SYMTAB]
   _NIL]

Create a Virtual-machine machine-state that contains a compiled-code pointer to the
open symboltable "TINY-DEFINITION-SYMTAB".

7_(_INT#LoadApply ms
   (][#TopLevelClosureObject 'EVAL_COM)
   < tinyprogram
      (][#TopLevelClosureObject 'final] >]

Prepare the machine to apply "EVAL_COM" to the command "tinyprogram" and
continuation "final".

8_(_INT#Break ms (][#ApplyBreak 'EVAL_COM) (][#ApplyBreak 'EVAL_EXPR)
   (][#ApplyBreak 'cont) (][#ApplyBreak 'econt]

Set breakpoints on calls to and returns from "EVAL_COM", "EVAL_EXPR", "cont" and
"econt".

9_Run ms
Broken after *BINDF* in EVAL_COM
   (Broken before EVAL_COM)
   _NIL
Begin running the initialized machine. Machine breaks upon entry to "EVAL_COM".

10_CF
EVAL_COM
com (WHEELE ((#BOOLEAN true) (#SEQ ((#SEQ ((#SEQ (& &))
 (#ASSIGN ("i" &)))) (#WHILE ((#NOT (#EQUAL &)) (#SEQ (& &)))))
 (#OUTPUT (#IDENT "res"))))))
cont ("final" : (INT#ClosureObject (final 1) NIL))
Dynamic chain: NIL
NIL

View the current frame of the machine. The values of the command "com" and
continuation "cont" are displayed. "com" is a "while" loop, and is the entire tiny
program.

11_Run
Broken after *BINDF* in EVAL_EXPR
(Broken before EVAL_EXPR)
T
Begin running again. Machine breaks upon entry to "EVAL_EXPR".

12_CF
EVAL_EXPR
expr ((#BOOLEAN true)
econt ("EVAL_COM.4.1.econt" : (INT#ClosureObject (EVAL_COM 142)
(INT#Frame EVAL_COM.4.1 & & &))
Dynamic chain: EVAL_COM.4
NIL

The expression being evaluated is the boolean expression "true", which is the boolean
expression of the outer "while" loop. The state is not required to evaluate it.

13_Run
Broken after *BINDF* in EVAL_COM.4.1.econt
(Broken before econt)
T
The value "true" is supplied to the expression continuation "econt" of step 12.
"econt" was passed by the "WHILE" "tcase" alternative of "EVAL_COM" to
"EVAL_EXPR".

14_CF
EVAL_COM.4.1.econt
value ((#BOOLEAN true)
Dynamic chain: EVAL_EXPR.2
NIL

15_Run
Broken after *BINDF* in EVAL_COM
(Broken before EVAL_COM)
T
Since "value" was "true", "EVAL_COM" should now be called with the statement part
of the "while" statement and a continuation that will evaluate the loop again. Since
AFDL + has call-by-value semantics; these arguments are first computed. The statement part is evaluated and pushed on the stack, and then "EVAL_COM" is called to produce the continuation. Unfortunately, this process repeats forever, as the following steps show.

16_CF
EVAL_COM
com ( WHILE ((BOOLEAN true) (SEQ ((SEQ ((SEQ (& &)))
ASSIGN (i &))) (WHILE ((NOT (EQUAL &)) (SEQ (& &)))))) (
OUTPUT IDENT "res"))
cont ("final" : (INT#ClosureObject (final 1) NIL))
Dynamic chain: EVAL_COM.4.1.econt
NIL

The invocation of "EVAL_COM" that must produce the continuation is entered.

17_Stk
I (SEQ ((SEQ ((SEQ ((SEQ (ASSIGN (num &)) (ASSIGN
(res &))) (ASSIGN (i (INTEGER 2))) (WHILE ((NOT (EQUAL
(IDENT i) (PLUS &))) (SEQ ((SEQ (& &)) (ASSIGN (i &)))))
(OUTPUT (IDENT res)))))))
NIL

The stack now contains the statement part of the "while" loop.

18_Run
Broken after *BINDF* in EVAL_EXPR
(Broken before EVAL_EXPR)
T

"EVAL_EXPR" is called again on the loop's boolean expression "true".

19_CF
EVAL_EXPR
expr (BOOLEAN true)
econt ("EVAL_COM.4.1.econt" : (INT#ClosureObject (EVAL_COM 142)
(INT#Frame EVAL_COM.4.1 & &)))
Dynamic chain: EVAL_COM.4
NIL

20_Run
Broken after *BINDF* in EVAL_COM.4.1.econt
(Broken before econt)
T

The expression continuation in "EVAL_COM" is called again with "true".

21_CF
EVAL_COM.4.1.econt
value (BOOLEAN true)
Dynamic chain: EVAL_EXPR.2
NIL

22_Run
Broken after *BINDF* in EVAL_COM
DEFINITIONS OF PROGRAMMING LANGUAGES

(Broken before EVAL_COM)

And yet another attempt is made to produce a continuation to the evaluation of the statement part.

23_CF
EVAL_COM
  com (WHILE ((BOOLEAN true) (SEQ ((SEQ ((SEQ (& &))
  (ASSIGN ("i" &))))) (WHILE ((NOT (#EQUAL &)) (SEQ (& &)))))) (OUTPUT
  (#IDENT "res")))
cont ("final": (INT#ClosureObject (final 1) NIL))
Dynamic chain: EVAL_COM.4.1.econt
NIL

24_Stk
1 (SEQ ((SEQ ((SEQ ((SEQ (ASSIGN ("num" &)) (ASSIGN
  ("res" &)))(ASSIGN ("i" (= INTEGER 2)))(WHILE ((NOT (#EQUAL
  (#IDENT "i") (#PLUS &)))) (SEQ ((SEQ (& &)) (ASSIGN (#IDENT ("i" &)))))))
  (#OUTPUT (#IDENT "res"))))
2 (SEQ ((SEQ ((SEQ ((SEQ (ASSIGN ("num" &)) (ASSIGN
  ("res" &)))(ASSIGN ("i" (= INTEGER 2)))(WHILE ((NOT (#EQUAL
  (#IDENT "i") (#PLUS &)))) (SEQ ((SEQ (& &)) (ASSIGN (#IDENT ("i" &)))))))
  (#OUTPUT (#IDENT "res"))))
NIL

The statement part of the "while" loop has now been pushed onto the stack twice.

25_DC
+ EVAL_COM
  + EVAL_COM.4.1.econt
  EVAL_EXPR.2
  + EVAL_EXPR
  EVAL_COM.4
  + EVAL_COM
  + EVAL_COM.4.1.econt
  EVAL_EXPR.2
  + EVAL_EXPR
  EVAL_COM.4
  + EVAL_COM
NIL

Display the dynamic chain. An obvious pattern can be seen, and would continue forever. The statement part of the "while" loop will never be evaluated, because its continuation can never be produced. Note that the "+" signs mark frames that have explicit dynamic chain pointers, i.e., were created by "call" instructions.

Transcript IV: Successful Interpretation
Tiny Definition, Version 3

The transcript of the application of the AFDL+ tools to Version 3, the final version of the Tiny definition, appears next. In this version the recursive continuation "EVAL_COM(com, cont)" of the "WHILE" alternative of "EVAL_COM" has been "shielded" by the function "new_cont", in order to simulate passing the continuation to "EVAL_COM" by name.
1. tc* TINY-DEFINITION-AS TINY-DEFINITION-SYMTAB

We use the tc* command here which is exactly equivalent to calling the function TCTypeCheck as in Transcripts II and III. This command is provided as a user convenience.

2. compile* TINY-DEFINITION-SYMTAB

update 21 (FUN (#MEMORY #VALUE #IDENT) #MEMORY)
value_of 7 (FUN (#MEMORY #IDENT) #VALUE_OR_UNBOUND)
error 6 (FUN (#STATE) #ANS)
final 6 (FUN (#STATE) #ANS)
initial_memory 6 (FUN (#IDENT) #VALUE_OR_UNBOUND)
EVAL_EXPR 242 (FUN (#EXPR #ECONT) #CONT)
EVAL_COM 210 (FUN (#COM #CONT) #CONT)

0 Warnings
0 Fatal Errors

TINY-DEFINITION-SYMTAB -- Compilation complete.
NIL

The definition is compiled. The integers in the second column are the number of virtual-machine instructions compiled for the corresponding AFDL object.

3. (NIL (tinystf (STFOpenFile 'TINY-TEST-AS.TXT])

Last updated 10-Jun-82 11:50:18
NIL

4. (NIL (tinyprogram (STFGetObject tinystf 'TinyProgram):1:1]
NIL

5. (NIL (ms (INT#CreateMachineState (SYMOpenTable 'TINY-DEFINITION-SYMTAB])
(ms reset)
NIL

6. (INT#LoadApply ms
   (INT#TopLevelClosureObject 'EVAL_COM)
   < tinyprogram
   (INT#TopLevelClosureObject 'final) >]
T

7. Run ms
   Halted
   NIL

Since no break points were set, the machine halts with a continuation on the top of the stack.

8. Stk

   ("EVAL_EXPR.4.1.cont" : (INT#ClosureObject (EVAL_EXPR 75)
   (INT#Frame EVAL_EXPR.4.1 & NIL & &)))
NIL

"EVAL_EXPR.4.1.cont" represents the return of the continuation "cont" by the "READ" tcase of "EVAL_EXPR", which corresponds to the factorial program's attempt to read the first datum from the input stream. Interpretation of the factorial program cannot
proceed until this continuation, suspended in a closure object, is applied to a machine state. This is done in the following steps.

9. (INT#LoadApply ms)
   (INT#PopVMSTACK ms)
   < < (INT#TopLevelClosureObject 'initial_memory)
   < '(#INTEGER 1) '(#INTEGER 2) '(#INTEGER 3) '(#INTEGER 4) > > ]

Prepare the machine to apply the continuation closure object that is on the top of the stack to an initial state. The "MEMORY" component of that state is the closure object "initial_memory". The "INPUT" component is a sequence of elements of type "VALUE", each of which was injected from type "INTEGER".

10_Stk
1  (("EVAL_EXPR.4.1.cont" : (INT#ClosureObject (EVAL_EXPR 75)
 (INT#Frame EVAL_EXPR.4.1 & NIL &))
2  ((("initial_memory" : (INT#ClosureObject (initial_memory 1) NIL))
 ((#INTEGER 1) (#INTEGER 2) (#INTEGER 3) (#INTEGER 4))))
NIL

View the stack before the continuation is applied to the initial state.

11. (INT#Break ms (INT#ApplyBreak 'update)(INT#ApplyBreak 'value_of))
T

Break the memory manipulation functions "update" and "value_of".

12_Run
Broken after *BINDF* in update
(Broken before update)
T

13_CF
update
   memory ("initial_memory" : (INT#ClosureObject (initial_memory 1) NIL))
   value (#INTEGER 1)
   id "num"

Dynamic chain: EVAL_COM.1.1.econt.1.new_cont
NIL

"update" is called to yield a new memory function that associates "id" with "value" (in this case "'num" with "(#INTEGER 1)") and otherwise calls "memory". "memory" is the "initial_memory" closure object.

Run
Broken before *RETURN* in update
(Broken after update)
T

Broken before the return from "update".

15_Stk
1  ("update.1.new_memory" : (INT#ClosureObject (update 5)
 (INT#Frame update.1 & NIL & &))
NIL

New memory function closure object is now on the stack; "'num" has been assigned the value "(#INTEGER 1)", which is the datum read from the input stream.
16_Run
Broken after *BINDF* in update
  (Broken before update)
T

17_CF
update
  memory  ("update.1.new_memory" : (INT#ClosureObject (update 5)
  (INT#Frame update.1 & NIL & &))
  value  (#INTEGER 1)
  id    "res"
Dynamic chain: EVAL_COM.1.1.econt.1.new_cont
NIL

  The "memory" argument of this invocation of "update" is now the closure object that
  was computed by the previous invocation of "update".

18_Run
Broken before *RETURN* in update
  (Broken after update)
T

  "res" has been assigned the (constant) value "(#INTEGER 1)".

19_Run
Broken after *BINDF* in update
  (Broken before update)
T

20_Run
Broken before *RETURN* in update
  (Broken after update)
T

  "i" has been assigned the (constant) value "(#INTEGER 2)".

21_Run
Broken after *BINDF* in value_of
  (Broken before value_of)
T

  The first memory access occurs.

22_CF
value_of
  memory  ("update.1.new_memory" : (INT#ClosureObject (update 5)
  (INT#Frame update.1 & NIL & &))
  id    "i"
Dynamic chain: EVAL_EXPR.3.1.cont.1
NIL

  The value of "i" is requested.

23_Run
Broken before *RETURN* in value_of
  (Broken after value_of)
T

24_Stk
  1
  (#VALUE (#INTEGER 2))
NIL
The value of "'i'" was "(# VALUE (# INTEGER 2))", of type "VALUE_OR_UNBOUND".

25.\(\text{INT}\#\text{UnBreak ms (INT}\#\text{ApplyBreak 'update]}\)

Unbreak "update".

26.\(\text{INT}\#\text{Break ms (INT}\#\text{ApplyBreak 'new_memory]}\)

Break "new_memory", which is the function of type "MEMORY" yielded by "update".

27.**Run**
Broken after *BINDF* in value_of
(Broken before value_of)

28.\(\text{CF}\)
value_of

memory ("update.1.new_memory" : (INT#ClosureObject (update 5)
(INT#frame update.1 & NIL & &))

id "num"

Dynamic chain: EVAL_EXPR.3.1.cont.1
NIL

The value of "'num'" is requested. The last value associated with "'num'" is "(# INTEGER 1)" (see steps 12 and 13).

29.**Run**
Broken after *BINDF* in update.1.new_memory
(Broken before new_memory)

"value_of" calls its argument "memory", which is an instance of "new_memory" yielded by "update".

30.**SC**
update.1.new_memory

ident "num"

update.1

new_memory

("update.1.new_memory" : (INT#ClosureObject (update 5)
(INT#frame update.1 & NIL & &)))

update

memory ("update.1.new_memory" : (INT#ClosureObject (update 5)
(INT#frame update.1 & NIL & &))

value (#INTEGER 2)

id "i"

NILD

View the static chain of "new_memory". "ident" is bound to "'num'", the identifier whose value is needed. This instance of the closure object "new_memory" is itself visible. The arguments to the invocation of "update" that created this instance of "new_memory" are also visible. "'i'" is associated with the value "(# INTEGER 2)". "memory" is the remainder of the memory that was supplied to that invocation of "update".
31_Run
Broken after *BINDF* in update.1.new_memory
   (Broken before new_memory)
   T
   Since "i" is not "num", "new_memory" calls "memory".

32_SC
update.1.new_memory
   ident  "num"

update.1
   new_memory
      ("update.1.new_memory" : (INT#ClosureObject (update 5)
                          (INT#Frame update.1 & NIL & &))

update
   memory ("update.1.new_memory" : (INT#ClosureObject (update 5)
                              (INT#Frame update.1 & NIL & &))
          value  (#INTEGER 1)
          id    "res"
   NIL

   This instance of "new_memory" knows about the value of "res".

33_Run
Broken after *BINDF* in update.1.new_memory
   (Broken before new_memory)
   T
   The next "deeper" instance of "new_memory" is called.

34_SC
update.1.new_memory
   ident  "num"

update.1
   new_memory
      ("update.1.new_memory" : (INT#ClosureObject (update 5)
                          (INT#Frame update.1 & NIL & &))

update
   memory ("initial_memory" : (INT#ClosureObject (initial_memory 1) NIL))
          value  (#INTEGER 1)
          id    "num"
   NIL

   This instance does "know about" the most recent binding of "num".
   and was yielded by the invocation of "update" that began in step 12.

35_Run
Broken before *RETURN* in update.1.new_memory
   (Broken after new_memory)
   T
   The value of "num" will now be yielded by each of the instances of "new_memory"
   that are active, and finally by "value_of" itself.

36_Stk
1
   (#VALUE (#INTEGER 1))
DEFINITIONS OF PROGRAMMING LANGUAGES

NIL

37_Run
Broken before *RETURN* in update.1.new_memory
(Broken after new_memory)
T

38_Stk
1 (#VALUE (#INTEGER 1))
NIL

39_Run
Broken before *RETURN* in update.1.new_memory
(Broken after new_memory)
T

40_Stk
1 (#VALUE (#INTEGER 1))
NIL

41_Run
Broken before *RETURN* in value_of
(Broken after value_of)
T

42_Stk
1 (#VALUE (#INTEGER 1))
NIL

The value of "'num'" is finally yielded by "value_of".

43_UBA
T

Remove all break points.

44_(INT#Break ms (INT#ApplyBreak 'EVALEXPR.4.1.cont])
T

Break the function "cont" of the "tcase" of "EVALEXPR". This function will be called with the "current" state each time a "read" expression is evaluated, which will occur on each successive traversal of the outer "while true" loop.

45_Run
Broken after *BINDF* in EVALEXPR.4.1.cont
(Broken before EVALEXPR.4.1.cont)
T

46_SC
EVALEXPR.4.1.cont
state
  (("update.1.new_memory" : (INT#ClosureObject (update 5)
  (INT#Frame update.1 & NIL & &)) )
  ((#INTEGER 2) (#INTEGER 3) (#INTEGER 4)))

EVALEXPR.4.1
cont
  ("EVALEXPR.4.1.cont" : (INT#ClosureObject (EVALEXPR 75)
  (INT#Frame EVALEXPR.4.1 & NIL & &)))

EVALEXPR.4
read
NIL
A "read" expression is evaluated (see the value of "expr") and the second element of the original input stream is about to be read. "econt" is the expression continuation that will be called with the value read, resulting in a continuation that will be called with the new state.

The "*" signs mark those frames that have explicit dynamic chain pointers, i.e., were created by "call" instructions. "while_cont" is the continuation that continues the computation after this traversal of the "while true" loop.

The third "read" occurs.

The results of computing the first two factorials are now on the stack. These values were pushed onto the stack by the "OUTPUT" alternative of "EVAL_COM" and, together with the third and fourth factorials, will be combined into a value of type "ANS", working from the top of the stack downward, when the input stream is exhausted and "error" is produced as the value of type "FINAL_ANSWER".
DEFINITIONS OF PROGRAMMING LANGUAGES

51_Run
Broken after *BIND* in EVAL_EXPR.4.1.cont
(Broken before EVAL_EXPR.4.1.cont)
T

52_CF
EVAL_EXPR.4.1.cont
state (("update.1.new_memory" : (INT#ClosureObject (update 5)
(INT#Frame update.1 & NIL & &)) ((#INTEGER 4)))
Dynamic chain: EVAL_COM.4.1.econt.1.while_cont
NIL

The fourth "read" occurs. The final integer will now be removed from the input stream.

53_Stk
1 (#INTEGER 6)
2 (#INTEGER 2)
3 (#INTEGER 1)
NIL

The third factorial has now been output, and is on the stack.

54_Run
Broken after *BIND* in EVAL_EXPR.4.1.cont
(Broken before EVAL_EXPR.4.1.cont)
T

55_CF
EVAL_EXPR.4.1.cont
state (("update.1.new_memory" : (INT#ClosureObject (update 5)
(INT#Frame update.1 & NIL & &)) NIL)
Dynamic chain: EVAL_COM.4.1.econt.1.while_cont
NIL

The final "read" occurs. An attempt to read past the end of the input stream will now occur.

56_Stk
1 (#INTEGER 24)
2 (#INTEGER 6)
3 (#INTEGER 2)
4 (#INTEGER 1)
NIL

The fourth and final factorial has now been output.

57_Run
Broken before *RETURN* in EVAL_EXPR.4.1.cont
(Broken after EVAL_EXPR.4.1.cont)
T

Since no input data remains, "cont" yields the "error" value of type "FINAL_ANSWER", which is then injected into type "ANS".

58_Stk
1 (#FINAL_ANSWER ~error)
2 (#INTEGER 24)
3 (#INTEGER 6)
4 (#INTEGER 2)
5 (#INTEGER 1)
NIL
Unbreak all functions.

Set a break point after all product denotation ("**PRDEN**") machine instructions. A break will thus occur each an instance of the "OUTPUT" alternative of "EVAL_COM" combines the second element of the stack (of type "VALUE") with the top element of the stack (of type "ANS") to form the new top element of the stack (of type "PARTIAL_ANSWER"; this value is then injected into type "ANS" and the process repeats).

Run Broken after **PRDEN** in EVAL_COM.2.1.econt.1.new_cont

Run Broken after **PRDEN** in EVAL_COM.2.1.econt.1.new_cont

Run Broken after **PRDEN** in EVAL_COM.2.1.econt.1.new_cont

Run Broken after **PRDEN** in EVAL_COM.2.1.econt.1.new_cont

Run Broken after **PRDEN** in EVAL_COM.2.1.econt.1.new_cont
DEFINITIONS OF PROGRAMMING LANGUAGES

69_DC
+ EVAL_COM.2.1.econt.1.new_cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_EXPR.3.1.cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_EXPR.3.1.cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_EXP.3.1.cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_EXPR.3.1.cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_EXPR.3.1.cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_EXPR.3.1.cont
  EVAL_EXPR.3.1.cont.1
+ EVAL_COM.1.1.econt.1.new_cont
+ EVAL_EXPR.4.1.cont
NIL

70_Run
Halted
T

The computation terminates.

71_Stk
1
  (#PARTIAL_ANSWER ((#INTEGER 1) (#PARTIAL_ANSWER ((#INTEGER 2)
    (#PARTIAL_ANSWER ((#INTEGER 6) (#PARTIAL_ANSWER ((#INTEGER 24)
      (#FINAL_ANSWER -error)))))))))
NIL

The resulting element of type "ANS" is on the top of the stack.

Transcript V: Optimization of Tail Recursion
Tiny Definition, Version 3

This transcript shows the effect of turning on the capability of the virtual machine to optimize tail recursion. This may be turned on or off at run time by setting the Lisp atom INT#OptimizeTailRecursion to T or NIL respectively. The optimization is expected to be useful in improving efficiency during interpretation of large semantic definitions.

We define a call to function X in context Y to be tail recursive if the value returned from the call to X is not modified before control returns from Y. Thus the return from X may as well return directly to Y's caller. The AFDL compiler detects all tail recursive calls in AFDL functions and emits special *CALL* instructions that behave like other *CALL* instructions until the flag INT#OptimizeTailRecursion is set at run time.

When tail recursion is being optimized it is possible for execution to continue, after a *RETURN* instruction, in a context that is not the normal target of the return. This may be somewhat confusing during "debugging." Hence by setting the INT#OptimizeTailRecursion flag the user is trading off clarity for debugging against increased execution speed.

In this transcript the machine state has already been initialized with the TINY semantic definition and the same test program written in TINY that was used in Transcript IV (see Steps 1 through 5 of Transcript IV). We first reset the machine state.

28_Reset ms
T

We prepare for the execution of the machine which is to apply EVAL_COM to the entire Tiny program and the continuation final.
29. (INT#LoadApply ms
    (INT#TopLevelClosureObject 'EVAL_COM)
    (tinyprogram (INT#TopLevelClosureObject 'final) >])

We request optimization of tail recursion by setting the atom
INT#OptimizeTailRecursion to a non-NIL value.

30. (SETQ INT#OptimizeTailRecursion T)
    (INT#OptimizeTailRecursion reset)

As a consequence of the call to INT#LoadApply we have a closure object embodying
the semantic function EVAL_COM on top of the stack. Just below it on the stack is the pair
of arguments to which it will be applied, namely the Tiny program and the closure object
that embodies the semantic function final.

31. Stk
    1  ("EVAL_COM": (INT#ClosureObject (EVAL_COM 1) NIL))
    2  ((#WHILE ((#BOOLEAN true) (#SEQ ((#SEQ ((#SEQ ((#SEQ &)) (#ASSIGN &)))
        (#WHILE ((#NOT &) (#SEQ &))) (#OUTPUT (#IDENT "res")))))
      ("final": (INT#ClosureObject (final 1) NIL)))
    NIL

We set a break around the semantic function EVAL_EXPR in order to view the dynamic
chain and its behavior when tail recursion is being optimized.

32. (INT#Break ms (INT#ApplyBreak 'EVAL_EXPR))

We run the machine through several breaks on EVAL_EXPR.

33. Run
    Broken after *BINDF* in EVAL_EXPR
    (Broken before EVAL_EXPR)

34. Run
    Broken after *BINDF* in EVAL_EXPR
    (Broken before EVAL_EXPR)

35. Run
    Broken before *RETURN* in EVAL_EXPR
    (Broken after EVAL_EXPR)

36. Run
    Broken after *BINDF* in EVAL_EXPR
    (Broken before EVAL_EXPR)

37. Run
    Broken after *BINDF* in EVAL_EXPR
    (Broken before EVAL_EXPR)

38. Run
    Broken after *BINDF* in EVAL_EXPR
    (Broken before EVAL_EXPR)

39. Run
    Broken before *RETURN* in EVAL_EXPR
    (Broken after EVAL_EXPR)
We are now in a break just before an incarnation of the semantic function EVAL_EXPR is due to execute its *RETURN* instruction. We observe the dynamic chain with the DC command. As before we have "+" signs that identify those frames that have explicit dynamic chain pointers. However, due to setting of the flag INT#OptimizeTailRecursion, we now also have "'"s marking those frames that were created by tail-recursive function calls. Executing a return from such a tail-recursive frame (i.e., one marked with a "'") causes control to return from the first frame "up" the dynamic chain that has an explicit dynamic chain pointer but is not tail-recursive (i.e., has a "+" but no "'"). Hence in the dynamic chain below, returning from the tail-recursive topmost frame EVAL_EXPR, as we are about to do, will cause execution to continue in the topmost frame named EVAL_COM.5.

```
40_DC
** EVAL_EXPR
  EVAL_EXPR.6
** EVAL_EXPR
  EVAL_EXPR.5
** EVAL_EXPR
  EVAL_COM.4
  EVAL_COM
  EVAL_COM.5
** EVAL_COM
  EVAL_COM.5
** EVAL_COM
  EVAL_COM.4.1.econt
    EVAL_EXPR.2
** EVAL_EXPR
  EVAL_COM.4
  EVAL_COM
NIL
```

We step the machine now through one instruction that causes it to execute the *RETURN* instruction it had broken before. The Step command (which may also be used with a positive integer argument) causes the machine to "single-step" through instructions. As each instruction is executed the instruction is printed out and, indented underneath it, is printed the value that is on top of the virtual-machine stack after the instruction execution. In this case we use the Step command without an argument implying that the current default machine state is to be stepped and the lack of an integer argument is taken to imply one step.

```
41_Step
  (*RETURN* EVAL_EXPR)
    (*"EVAL_EXPR.3.1.cont" : (INT#ClosureObject
      (EVAL_EXPR 35) (INT#Frame EVAL_EXPR.3.1 & NIL & &)))
T
```

As predicted above, the return from the tail-recursive EVAL_EXPR frame causes execution to return to the frame EVAL_COM.5.

```
42_DC
  EVAL_COM.5
  ** EVAL_COM
    EVAL_COM.5
  ** EVAL_COM
```
We remove all breaks on the machine and let it run until it halts.

As a result of applying EVAL_COM to the Tiny program and the continuation final we get a continuation deposited on top of the virtual-machine stack. This continuation (see TINY definition) expects an object of type STATE as argument and will provide an object of type ANS as result.

We get ready to apply the just-computed continuation to a Tiny program state that is the initial memory and an input sequence consisting of the integer 4. We expect the factorial of 4 as the result of the next machine execution.

We introduce a break to allow us to see the dynamic chain at a convenient point for purposes of illustration.

We are now poised to execute a *RETURN* instruction from the tail-recursive frame EVAL_COM.2.1.econt.1.new_cont but note that all frames further "up" the dynamic chain (except the one at the bottom of the list below) are tail-recursive as well. Thus execution of this return should leave the dynamic chain empty.
DEFINITIONS OF PROGRAMMING LANGUAGES

We execute the *RETURN* with a Step command and note that the stack now contains the appropriate result.

50_Step
  (*RETURN* EVAL_COM.2.1.econt.1.new_cont)
  (#PARTIAL_ANSWER ((#INTEGER 24) (#FINAL_ANSWER ~error)))
T
51_DC
NIL

As expected, the dynamic chain is now completely clear and the next Run command will simply cause the machine to halt.

52_Run
Halted
T

The expected object of type ANS is left on top of the virtual-machine stack.

53_Stk
1
  (#PARTIAL_ANSWER ((#INTEGER 24) (#FINAL_ANSWER ~error)))
NIL
I. ISI EXTENSIONS TO AFDL

We briefly describe here the differences between AFDL and our upward compatible extension AFDL +. It is assumed that the reader is reasonably familiar with AFDL and the Ada FSD.

AFDL is a purely applicative language composed of a few Ada constructs such as if-then-else, case, blocks, variable declarations, function subprograms, etc. but no assignment. The major difference between the use of these constructs in Ada and in AFDL is that in AFDL functions may be passed as arguments to other functions. This is necessary for writing denotational style semantic definitions. Accordingly, the types of functional objects may be defined with function type declarations. Furthermore, in AFDL the only possible mode of function arguments is similar to the Ada in mode (i.e., parameter passing by value). This is a fundamental divergence from the lambda calculus and must be duly recognized when writing semantic definitions in AFDL.

In extending AFDL to produce AFDL + we introduced three further type declarations, ten domain operations and one compound expression construct. In addition, we have made AFDL + a pure expression language thereby rendering the return keyword and construct optional. In order to facilitate programming denotational semantics, AFDL + function calls may be curried.

If the sum type $T$ is declared via

$$\text{sum type } T(T_1, \ldots, T_n),$$

where $n \geq 2$ and all $T_i$ are unique names, then $x$ inj $T$ injects an element of component type $T_i$ into the sum type $T$; $x$ elt $T_i$ tests whether an element $x$ of sum type $T$ was injected from component type $T_i$; $x$ pro $T_i$ projects an element $x$ of sum type $T$ into component type $T_i$ if $x$ elt $T_i$ is true, and causes a fatal error in the AFDL + virtual machine otherwise.

If the product type $T$ is declared via

$$\text{product type } T(T_1, \ldots, T_n),$$

where $n \geq 2$, then an element of $T$ is constructed from elements $x_i$ of type $T_i$ by the tupling operation $(x_1, \ldots, x_n)$. The $i$-th component of a value $x$ of product type $T$ is selected by the operation $x:i$, where $i$ must be a constant (literal).

A sequence type $T$ with components of type $S$ is declared via

$$\text{sequence type } T S.$$  

An element of sequence type $T$ is constructed from elements $x_i$ of type $S$ by the operation $[x_1, \ldots, x_n]$, $n \geq 0$. $[ ]$ denotes the empty element of any sequence type. The $i$-th component (head) of an element $x$ of sequence type $T$ is selected by the operation $x:i$, where $i$ may be any integer expression. The $i$-th tail of $x = [x_1, \ldots, x_n]$ is selected by the operation $x::i$, and is equal to $[x_{i+1}, \ldots, x_n]$ (if $i = n$ then $x::i$ is equal to $[ ]$). If $i$ is "out of range" ($\leq 0$ or $> n$) for $: :$ or $::$, then a fatal error occurs in the AFDL + virtual machine. Two elements of a given sequence type may be concatenated using the binary concatenation operation $\&$. The length of an element of a sequence type may be obtained via the unary operation $\text{length}$.

A type $T$ may be declared equivalent to a type $S$ by the declaration

$$\text{type } T \text{ is } S.$$
This permits an element $x$ of type $T$ to be converted to type $S$ by applying the cast $S(x)$; conversely, an element $y$ of type $S$ can be converted to type $T$ via the cast $T(y)$. Type conversions must be explicit and have no run-time significance. Equivalent types provide a way of declaring a sum type $T$ with multiple components $T_i$ that are equivalent to some type $S$. If $x : T_i$, then $S(x \text{ pro } T_i)$ is of type $S$, and if $y$ is an element of type $S$, then $(T_i(y)) \text{ inj } T$ is an element of type $T$.

The AFDL $+$ type case expression is a variant of the usual case construct and is provided to simplify the manipulation of sum types. The tcase expression

```
tcase expr is
  when $t_1 : T_1$ => expr$_1$
  ...
  when $t_n : T_n$ => expr$_n$
  when others $t : T$ => default
end tcase
```

evaluates $expr_i$ in a new scope in which $t_i$ is bound to $expr \text{ pro } T_i$ if $expr \text{ elt } T_i$, and yields the resulting value as the value of the type case expression. If a when clause for the appropriate $T_i$ is not provided, then if the others clause is present, "default" is evaluated in a scope in which $t$ is bound to $expr$, and the resulting value is yielded as the value of the type case expression; otherwise, a fatal error occurs in the AFDL $+$ virtual machine.
package TINY is

-- Syntactic Domains

sum type EXPR (INTEGER, BOOLEAN, IDENT, READ, NOT, EQUAL, PLUS);

type IDENT is private;
type READ is private;
type NOT is EXPR;
product type EQUAL (EXPR, EXPR);
product type PLUS (EXPR, EXPR);

sum type COM (ASSIGN, OUTPUT, IF, WHILE, SEQ);

product type ASSIGN (IDENT, EXPR);
type OUTPUT is EXPR;
product type IF (EXPR, COM, COM);
product type WHILE (EXPR, COM);
product type SEQ (COM, COM);

-- Semantic Domains

product type STATE (MEMORY, INPUT);

function type MEMORY (id: IDENT) return VALUE_OR_UNBOUND;
sum type VALUE_OR_UNBOUND (VALUE, UNBOUND_VALUE);
type UNBOUND_VALUE is (unbound);

sequence type INPUT VALUE;

sum type VALUE (INTEGER, BOOLEAN);

function type CONT (state: STATE) return ANS;
function type ECONT (value: VALUE) return CONT;

sum type ANS (FINAL_ANSWER, PARTIAL_ANSWER);
type FINAL_ANSWER is (error, stop);
product type PARTIAL_ANSWER (VALUE, ANS);

-- Auxiliary Functions

function update (memory: MEMORY; value: VALUE; id: IDENT) return MEMORY is
  function new_memory (ident: IDENT) return VALUE_OR_UNBOUND is
    begin
      if ident = id then value inj VALUE_OR_UNBOUND
    end
DEFINITIONS OF PROGRAMMING LANGUAGES

else memory(ident) end if
end new_memory;
begin new_memory end update;

function value_of (memory: MEMORY; id: IDENT) return VALUE_OR_UNBOUND is
begin memory(id) end value_of;

function error (state: STATE) return ANS is
begin error inj ANS end error;

function final (state: STATE) return ANS is
begin stop inj ANS end final;

function initial_memory (id: IDENT) return VALUE_OR_UNBOUND is
begin unbound inj VALUE_OR_UNBOUND end initial_memory;

-- Semantic Functions

function EVAL_EXPR (expr: EXPR; econt: ECONT) return CONT is begin
tcase expr is
when integer: INTEGER =⇒ econt(integer inj VALUE);
when boolean: BOOLEAN =⇒ econt(boolean inj VALUE);
when ident: IDENT =⇒
  declare function cont (state: STATE) return ANS is
    value: VALUE_OR_UNBOUND =⇒ value_otlstate:1, ident);
    begin if value elt UNBOUND_VALUE then error inj ANS
    else econt(value pro VALUE)(state) end if end cont;

begin cont end;
when read: READ =⇒
  declare function cont (state: STATE) return ANS is
    begin if state:2 = [] then error inj ANS
    else econt(state:2:1)((state:1, state:2::1)) end if end cont;

begin cont end;
when not: NOT =⇒
  EVAL_EXPR( EXPR(not),
    declare function new_econt (value: VALUE) return CONT is
    begin
      if value elt BOOLEAN then econt((not (value pro BOOLEAN)) inj VALUE)
      else error end if
    end new_econt;

begin new_econt end);
when equal: EQUAL =⇒
  EVAL_EXPR
  ( equal:1,
    declare function econt1 (value1: VALUE) return CONT is
    begin
      EVAL_EXPR
      ( equal:2,
        declare function econt2 (value2: VALUE) return CONT is


begin econt((value1 = value2) inj VALUE) end econt2;
begin econt2 end)
end econt1;

when plus: PLUS =>
EVAL_EXPR
( plus:1,
define function econt1 (value1: VALUE) return CONT is
begin
EVAL_EXPR
( plus:2,
define function econt2 (value2: VALUE) return CONT is
begin
if value1 elt INTEGER and value2 elt INTEGER
then econt( ((value1 pro INTEGER) + (value2 pro INTEGER)) inj VALUE)
else error end if
end econt2;
begin econt2 end)
end econt1;
begin econt end)
end EVAL_EXPR;

function EVAL_COM (com: COM; cont: CONT) return CONT is
begin
tcase com is
when ass: ASSIGN =>
EVAL_EXPR
( ass:2,
define function econt (value: VALUE) return CONT is
function new_cont (state: STATE) return ANS is
begin
(value, cont(state)) inj ANS end new_cont;
begin new_cont end econt;
begin econt end);
when out: OUTPUT =>
EVAL_EXPR
( out:4

define function econt (value: VALUE) return CONT is
function new_cont (state: STATE) return ANS is
begin
(value, cont(state)) inj ANS end new_cont;
begin new_cont end econt;
begin econt end);
when if: IF =>
EVAL_EXPR
( if:1,
define function econt (value: VALUE) return CONT is
begin
if value elt BOOLEAN then
if value pro BOOLEAN then EVAL_COM(if:2, cont)
else EVAL_COM(if:3, cont) end if
else error end if
end econt;

begin econt end);
when while: WHILE =>
define function new_cont (state: STATE) return ANS is
begin
EVAL_EXPR
( while:1,
define function econt (value: VALUE) return CONT is
begin
if value elt BOOLEAN then
if value pro BOOLEAN
then EVAL_COM(while:2, EVAL_COM(com, cont))
else cont end if
else error end if
end econt;
begin econt end)
(state); -- Curried call
end new_cont;
begin new_cont end;
when seq: SEQ =>
EVAL_COM(seq:1, EVAL_COM(seq:2, cont))
end tcase
end EVAL_COM;
end TINY;
III. AFDL+ ABSTRACT SYNTAX

AFDL Abstract Syntax (List Representation)
D. Martin and A. Stoughton
17 Feb. 1982

program: comp_unit_s

comp_unit: fun_decl
       | fun_spec
       | pkg_spec
       | pkg_body

pkg_spec: (PSPEC pkg_id decl_item_s)
pkg_body: (PBODY pkg_id decl_item_s)

decl_item: var_spec
       | var_decl
       | fun_spec
       | fun_decl
       | type_decl
       | fun_type_decl
       | prod_type_decl
       | sum_type_decl
       | seq_type_decl
       | pkg_spec
       | pkg_body
       | use_clause

var_spec: (VSPEC var_id_s type_id)
type_decl: (TDECL type_id type_def)
type_def: PRIVATE
       | enum_type
       | type_id
       | -- private type
       | -- enumerated type
       | -- type equivalence

enum_type: const_id_s
       | var_id_s

fun_type_decl: (FNTDECL type_id fun_type)
prod_type_decl: (PRTDECL type_id type_id_s)
sum_type_decl: (SMTDECL type_id type_id_s)
DEFINITIONS OF PROGRAMMING LANGUAGES

seq_type_decl: (SQDECL type_id type_id)

use_clause: (USE pkg_id_s)

fun_decl: (FDECL fun_id fun_type fun_def)

fun_spec: (FSPEC fun_id fun_type)

fun_type: (MAP type_id_s type_id)

fun_def: (LAM id_s expr)

eexpr: (LET decl_s expr)
| (APPLY expr expr_s)
| (IF expr expr expr)
| (CASE expr alt_s)
| (TCASe expr t_alt_s)
| (WARNING expr msg)
| simple_expr

decl: var_decl
| fun_spec
| fun_decl

var_decl: (VDECL var_id type_id expr)

alt: (choice_s expr)

choice: id
| OTHERS

t_alt: (ALT var_id type_id expr)
| (OTHERS var_id type_id expr)

simple_expr: (binop expr expr)
| (unop expr)
| (ELT expr type_id)
| (INJ expr type_id)
| (INJ expr type_id (FROM type_id))
| (PRO expr type_id)
| (PRDEN expr_s)
| (SQDEN expr_s)
| (CAST type_id expr)
| id
| integer
| string

id: var_id
| fun_id
| const_id
| type_id
| binop:   | AND  |
|         | OR   |
|         | XOR  |
|         | ANDTHEN |
|         | ORELS |}

| unop:   | NOT  |
|         | UMINUS |
|         | LENGTH |
IV. AFDL + CONCRETE AND ABSTRACT SYNTAX

*** OUTPUT FROM GRAMMAR ANALYSIS PROGRAM LR for the AFDL + GRAMMAR ***

<table>
<thead>
<tr>
<th>TERMINAL</th>
<th>NON TERMINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>1 (IRHN LI2)</td>
</tr>
<tr>
<td>(</td>
<td>2 TYPm ID List&gt;</td>
</tr>
<tr>
<td>)</td>
<td>3 )</td>
</tr>
<tr>
<td>*</td>
<td>4 (XPR1s)</td>
</tr>
<tr>
<td>+</td>
<td>5 (XPR1s)</td>
</tr>
<tr>
<td>.</td>
<td>6 (XPR1s)</td>
</tr>
<tr>
<td>,</td>
<td>7 (XPR1s)</td>
</tr>
<tr>
<td>etconst</td>
<td>8 (XPR1s)</td>
</tr>
<tr>
<td>integer</td>
<td>9 (XPR1s)</td>
</tr>
<tr>
<td>. procname</td>
<td>10 (XPR1s)</td>
</tr>
<tr>
<td>. string</td>
<td>11 (XPR1s)</td>
</tr>
<tr>
<td>. typemark</td>
<td>12 (XPR1s)</td>
</tr>
<tr>
<td>. varname</td>
<td>13 (XPR1s)</td>
</tr>
<tr>
<td>/</td>
<td>14 (NUMERATED TYPE DEFINITION)</td>
</tr>
<tr>
<td>:</td>
<td>15 (XPR1s)</td>
</tr>
<tr>
<td>;</td>
<td>16 (XPR1s)</td>
</tr>
<tr>
<td>=</td>
<td>17 (XPR1s)</td>
</tr>
<tr>
<td>= =</td>
<td>18 (XPR1s)</td>
</tr>
<tr>
<td>&lt;</td>
<td>19 (XPR1s)</td>
</tr>
<tr>
<td>&gt;</td>
<td>20 (XPR1s)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>21 (XPR1s)</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>22 (XPR1s)</td>
</tr>
<tr>
<td>&gt; =</td>
<td>23 (XPR1s)</td>
</tr>
<tr>
<td>&gt; &gt;</td>
<td>24 (XPR1s)</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>25 (XPR1s)</td>
</tr>
<tr>
<td>AND</td>
<td>26 (XPR1s)</td>
</tr>
<tr>
<td>BEGIN</td>
<td>27 (XPR1s)</td>
</tr>
<tr>
<td>BODY</td>
<td>28 (XPR1s)</td>
</tr>
<tr>
<td>CASE</td>
<td>29 (XPR1s)</td>
</tr>
<tr>
<td>DECLAR</td>
<td>30 (XPR1s)</td>
</tr>
<tr>
<td>ELSE</td>
<td>31 (XPR1s)</td>
</tr>
<tr>
<td>END</td>
<td>32 (XPR1s)</td>
</tr>
<tr>
<td>IF</td>
<td>33 (XPR1s)</td>
</tr>
<tr>
<td>IM</td>
<td>34 (XPR1s)</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>35 (XPR1s)</td>
</tr>
<tr>
<td>IF</td>
<td>36 (XPR1s)</td>
</tr>
<tr>
<td>INJ</td>
<td>37 (XPR1s)</td>
</tr>
<tr>
<td>IS</td>
<td>38 (XPR1s)</td>
</tr>
<tr>
<td>IINGHT</td>
<td>39 (XPR1s)</td>
</tr>
<tr>
<td>NOT</td>
<td>40 (XPR1s)</td>
</tr>
<tr>
<td>OR</td>
<td>41 (XPR1s)</td>
</tr>
<tr>
<td>OTHERS</td>
<td>42 (XPR1s)</td>
</tr>
<tr>
<td>PACKAGE</td>
<td>43 (XPR1s)</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>44 (XPR1s)</td>
</tr>
<tr>
<td>PROC</td>
<td>45 (XPR1s)</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>46 (XPR1s)</td>
</tr>
<tr>
<td>RETURN</td>
<td>47 (XPR1s)</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>48 (XPR1s)</td>
</tr>
<tr>
<td>SUM</td>
<td>49 (XPR1s)</td>
</tr>
<tr>
<td>ICAS</td>
<td>50 (XPR1s)</td>
</tr>
<tr>
<td>IH</td>
<td>51 (XPR1s)</td>
</tr>
<tr>
<td>IPI</td>
<td>52 (XPR1s)</td>
</tr>
<tr>
<td>USE</td>
<td>53 (XPR1s)</td>
</tr>
<tr>
<td>WHEN</td>
<td>54 (XPR1s)</td>
</tr>
<tr>
<td>XOR</td>
<td>55 (XPR1s)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>57 (XPR1s)</td>
</tr>
<tr>
<td>8</td>
<td>58 (XPR1s)</td>
</tr>
<tr>
<td>9</td>
<td>59 (XPR1s)</td>
</tr>
<tr>
<td>10</td>
<td>60 (XPR1s)</td>
</tr>
<tr>
<td>11</td>
<td>61 (XPR1s)</td>
</tr>
<tr>
<td>12</td>
<td>62 (XPR1s)</td>
</tr>
<tr>
<td>13</td>
<td>63 (XPR1s)</td>
</tr>
<tr>
<td>14</td>
<td>64 (XPR1s)</td>
</tr>
<tr>
<td>15</td>
<td>65 (XPR1s)</td>
</tr>
<tr>
<td>16</td>
<td>66 (XPR1s)</td>
</tr>
<tr>
<td>17</td>
<td>67 (XPR1s)</td>
</tr>
<tr>
<td>18</td>
<td>68 (XPR1s)</td>
</tr>
<tr>
<td>19</td>
<td>69 (XPR1s)</td>
</tr>
<tr>
<td>20</td>
<td>70 (XPR1s)</td>
</tr>
<tr>
<td>21</td>
<td>71 (XPR1s)</td>
</tr>
<tr>
<td>22</td>
<td>72 (XPR1s)</td>
</tr>
<tr>
<td>23</td>
<td>73 (XPR1s)</td>
</tr>
<tr>
<td>24</td>
<td>74 (XPR1s)</td>
</tr>
<tr>
<td>25</td>
<td>75 (XPR1s)</td>
</tr>
<tr>
<td>26</td>
<td>76 (XPR1s)</td>
</tr>
<tr>
<td>27</td>
<td>77 (XPR1s)</td>
</tr>
<tr>
<td>28</td>
<td>78 (XPR1s)</td>
</tr>
<tr>
<td>29</td>
<td>79 (XPR1s)</td>
</tr>
<tr>
<td>30</td>
<td>80 (XPR1s)</td>
</tr>
<tr>
<td>31</td>
<td>81 (XPR1s)</td>
</tr>
<tr>
<td>32</td>
<td>82 (XPR1s)</td>
</tr>
<tr>
<td>33</td>
<td>83 (XPR1s)</td>
</tr>
<tr>
<td>34</td>
<td>84 (XPR1s)</td>
</tr>
<tr>
<td>35</td>
<td>85 (XPR1s)</td>
</tr>
<tr>
<td>36</td>
<td>86 (XPR1s)</td>
</tr>
<tr>
<td>37</td>
<td>87 (XPR1s)</td>
</tr>
<tr>
<td>38</td>
<td>88 (XPR1s)</td>
</tr>
<tr>
<td>39</td>
<td>89 (XPR1s)</td>
</tr>
<tr>
<td>40</td>
<td>90 (XPR1s)</td>
</tr>
<tr>
<td>41</td>
<td>91 (XPR1s)</td>
</tr>
<tr>
<td>42</td>
<td>92 (XPR1s)</td>
</tr>
<tr>
<td>43</td>
<td>93 (XPR1s)</td>
</tr>
<tr>
<td>44</td>
<td>94 (XPR1s)</td>
</tr>
<tr>
<td>45</td>
<td>95 (XPR1s)</td>
</tr>
<tr>
<td>46</td>
<td>96 (XPR1s)</td>
</tr>
<tr>
<td>47</td>
<td>97 (XPR1s)</td>
</tr>
<tr>
<td>48</td>
<td>98 (XPR1s)</td>
</tr>
<tr>
<td>49</td>
<td>99 (XPR1s)</td>
</tr>
<tr>
<td>50</td>
<td>100 (XPR1s)</td>
</tr>
<tr>
<td>51</td>
<td>101 (XPR1s)</td>
</tr>
<tr>
<td>52</td>
<td>102 (XPR1s)</td>
</tr>
<tr>
<td>53</td>
<td>103 (XPR1s)</td>
</tr>
<tr>
<td>54</td>
<td>104 (XPR1s)</td>
</tr>
<tr>
<td>55</td>
<td>105 (XPR1s)</td>
</tr>
<tr>
<td>56</td>
<td>106 (XPR1s)</td>
</tr>
<tr>
<td>57</td>
<td>107 (XPR1s)</td>
</tr>
<tr>
<td>58</td>
<td>108 (XPR1s)</td>
</tr>
<tr>
<td>59</td>
<td>109 (XPR1s)</td>
</tr>
<tr>
<td>60</td>
<td>110 (XPR1s)</td>
</tr>
<tr>
<td>61</td>
<td>111 (XPR1s)</td>
</tr>
<tr>
<td>62</td>
<td>112 (XPR1s)</td>
</tr>
<tr>
<td>63</td>
<td>113 (XPR1s)</td>
</tr>
<tr>
<td>64</td>
<td>114 (XPR1s)</td>
</tr>
<tr>
<td>65</td>
<td>115 (XPR1s)</td>
</tr>
<tr>
<td>66</td>
<td>116 (XPR1s)</td>
</tr>
<tr>
<td></td>
<td>&lt;SIMPLE EXPRESSION&gt;</td>
</tr>
</tbody>
</table>
DEFINITIONS OF PROGRAMMING LANGUAGES

THI PRODUCTIONS

1 <SYSTEM GOAL SYMBOL> ::= "*" <AFDL_PROGRAM> "*"

--- LR(1) Grammar for Ada Formal Definition Language (AFDL)
--- This version includes Appendices G, H, I, J, and K of the Ada ISD
--- as well as the static and dynamic functions of Chapters 3, 4, 5, 6.
--- 7, 8, 9, 10, 11, and 12.

2 <AFDL_PROGRAM> ::= <COMPILATION_UNIT_LIST> S | S10 AbstractSyntaxTrees STAR:1]

3 <COMPILATION_UNIT_LIST> ::= <COMPILATION_UNIT_LIST> <COMPILATION_UNIT> ;

4 | <COMPILATION_UNIT> ; S | [PRINIDI STAR:1:2] [PRPRI] <>

5 <COMPILATION_UNIT> ::= <FUNCTION DECLARATION>

6 | <PACKAGE SPECIFICATION>

7 | <PACKAGE BODY>

---

************ APPENDIXES G through K ************

8 <PACKAGE SPECIFICATION> ::= PACKAGE .procname IS <DECLARATIVE ITEM LIST> END <FUNCTION NAME OPTION>

S > - :PSPI12

9 <PACKAGE BODY> ::= PACKAGE BODY .procname IS <DECLARATIVE PART> END <FUNCTION NAME OPTION>

S > - :PRODI12

10 <DECLARATIVE ITEM LIST> ::= <DECLARATIVE ITEM> ; S >

11 | <DECLARATIVE ITEM LIST> <DECLARATIVE ITEM> ; S >

12 <DECLARATIVE PART> ::= <DECLARATIVE ITEM> ; S >

13 | <PACKAGE BODY> ; S >

14 | <DECLARATIVE PART> <DECLARATIVE ITEM> ; S >

15 | <DECLARATIVE PART> <PACKAGE BODY> ; S >

16 <DECLARATIVE ITEM> ::= <VARIABLE_SPECIFICATION>

17 | <VARIABLE_DECLARATION>

18 | <FUNCTION_SPECIFICATION> S > SWAP[1 2] -> :PSPI12

19 | <FUNCTION_DECLARATION>

20 | <TYPE_DECLARATION>

21 | <FUNCTION_TYPE_DECLARATION>

22 | <PRODUCT_TYPE_DECLARATION>

23 | <SUM_TYPE_DECLARATION>

24 | <SEQUENCE_TYPE_DECLARATION>

25 | <PACKAGE_SPECIFICATION>

26 | <USE_CLAUSE>


28 | <2_VAR_ID LIST> : <TYPE_ID> S > :VSPIC12

29 <TYPE DECLARATION> ::= <TYPE <TYPE ID> IS <TYPE DEFINITION> S > :IDC12

30 <TYPE DEFINITION> ::= (<NUMERATED TYPE DEFINITION>

31 | PRIVATE S > LOAD(PRIVATE)

32 | <TYPE_ID> -- type "equivalence"

33 <NUMERATED TYPE DEFINITION> ::= <CONST_ID LIST>

34 | <VAR_ID LIST>

35 <CONST_ID_LIST> ::= <CONST_ID> ; S >

36 | <VAR_ID_LIST> , <CONST_ID> S >

37 <VAR_ID_LIST> ::= varname ; S >

38 | <VAR_ID_LIST> , varname S >
TOOLS FOR TESTING DENOTATIONAL SEMANTIC

39 <2 VAR ID LIST> ::= .varname , .varname S> MAXIUP1[2]
40 | <2 VAR ID LIST> , .varname S> <<

41 <2 TYPE ID LIST> ::= <TYPE ID> , <TYPE ID> S> MAXIUP1[2]
42 | <2 TYPE ID LIST> , <TYPE ID> S> <<

43 <FUNCTION TYPE DECLARATION> ::= FUNCTION TYPE <TYPE ID> ( <PARAMETER DECLARATION LIST> ) <RETURN TYPE>

45 <PRODUCT TYPE DECLARATION> ::= PRODUCT TYPE <TYPE ID> ( <2 TYPE ID LIST> )
46 | <PRODUCT TYPE DECLARATION> S> : PRDICT12

47 <SUM TYPE DECLARATION> ::= SUM TYPE <TYPE ID> ( <2 TYPE ID LIST> ) S> : SMIDICI12

48 <SEQUENCE TYPE DECLARATION> ::= SEQUENCE TYPE <TYPE ID> <TYPE ID> S> : SQIDICI12

49 <USI CLAUSE> ::= USI <PKG ID LIST> S> : US111

50 <FUNCTION DECLARATION> ::= <FUNCTION SPECIFICATION> IS <FUNCTION BODY>
51 | <FUNCTION SPECIFICATION> IS ( NULL BLOCK BODY) S> SWAP[1 2] - : SPFC12

52 <FUNCTION SPECIFICATION> ::= FUNCTION <ID> <FORMAL PAR1> <RETURN TYPE>
53 | <FUNCTION SPECIFICATION> S> : MAP12

54 <FORMAL PAR1> ::= S> LOAD11111 | -- IMPI1Y
55 | ( <PARAMETER DECLARATION LIST> ) S> BSUS1111 | -- burst parameter list

56 <PARAMETER DECLARATION LIST> ::= <PARAMETER DECLARATION LIST> , <PARAMETER DECLARATION>
57 | <PARAMETER DECLARATION> S> <<

58 <PARAMETER DECLARATION> ::= <NAME LIST> : <TYPE ID> S> MAXIUP1[2]

59 <RETURN TYPE> ::= RETURN <TYPE ID>

60 <NULL BLOCK BODY> ::= BEGIN END <FUNCTION NAMI OPTION> S> | -- for incomplete procedures

61 <FUNCTION BODY> ::= <DECLARATION LIST> <BLOCK BODY> S> ::111

62 <DECLARATION LIST> ::= <DECLARATION LIST> , <DECLARATION> S> <<
63 | <DECLARATION> S> <<

64 <DECLARATION> ::= <VARIABLE DECLARATION>
65 | <FUNCTION DECLARATION>
66 | <FUNCTION SPECIFICATION> S> SWAP[1 2] - : SPFC12

67 <VARIABLE DECLARATION> ::= .varname : <TYPE ID> : <EXPRESSION> S> : VDICI13

68 <NAME LIST> ::= <NAME LIST> , <NAME> S> <<
69 | <NAME> S> <<

70 <NAME> ::= .varname
71 | .procname

72 <CONST ID> ::= .etconst S> : REPLACA STAR (PACK* " STAR:1))
73 <TYPE ID> ::= .typemark S> : REPLACA STAR (PACK* " STAR:1))
DEFINITIONS OF PROGRAMMING LANGUAGES

74 <EXPRESSION> ::= <RETURN_EXPR>
75 | <RELATION>
76 | <AND_EXPRESSION>
77 | <OR_EXPRESSION>
78 | <XOR_EXPRESSION>
79 | <ANDTHEN_EXPRESSION>
80 | <ORELSE_EXPRESSION>

81 <SEMI> ::= S> -- <EMPTY> &C optional semicolon
     | -- discard stacked NIL
     | -- nothing stacked here

83 <RETURN_EXPR> ::= RETURN <EXPRESSION>

84 <IF_EXPR> ::= IF <CONDITION> THEN <EXPRESSION> <SEMI> <ELSE_EXPR> END IF

85 <CONDITION> ::= <EXPRESSION>

86 <ELSE_EXPR> ::= ELSE <EXPRESSION> <SEMI> END IF

88 <CASE_EXPR> ::= CASE <EXPRESSION> IS <ALTERNATIVE_LIST> END CASE

89 <ALTERNATIVE_LIST> ::= <ALTERNATIVE_LIST> <ALTERNATIVE>
90 | <ALTERNATIVE>

91 <ALTERNATIVE> ::= WHEN <CHOICE_LIST> => <EXPRESSION> <SEMI> MAKEUPI[2]

92 <CHOICE_LIST> ::= <CHOICE_LIST> | <CHOICE>
93 | <CHOICE>

94 <CHOICE> ::= <CONST_ID>
95 | varname
96 | OTHERS S> LOAD[OTHERS]

97 <TYPE_CASE_EXPR> ::= ICASE <EXPRESSION> IS <TYPE_ALT_LIST> END ICASE

98 <TYPE_ALT_LIST> ::= <TYPE_ALT>
99 | <TYPE_ALT_LIST> <TYPE_ALT>

100 <TYPE_ALT> ::= WHEN .varname : <TYPE_ID> => <EXPRESSION> <SEMI> MAKEUPI[4]

102 <WHEN_OTHERS> ::= WHEN OTHERS S> LOAD[OTHERS]

103 <TYPE_COERCION> ::= <TYPE_ID> ( <EXPRESSION> ) S> CASE12

104 <BLOCK_BODY> ::= BEGIN <DECLARATION_LIST> <BLOCK_BODY> S> :CASE12

105 | <BLOCK_BODY>

106 <BLOCK_BODY> ::= BEGIN <EXPRESSION> <SEMI> END <FUNCTION_NAME_OPTION>

107 <FUNCTION_NAME_OPTION> ::= -- <IMPLIED>

108 | <ID>

109 <AND_EXPRESSION> ::= <RELATION> AND <RELATION> S> :AND12

110 | <AND_EXPRESSION> AND <RELATION> S> :AND12

111 <OR_EXPRESSION> ::= <RELATION> OR <RELATION> S> :OR12

112 | <OR_EXPRESSION> OR <RELATION> S> :OR12

113 <XOR_EXPRESSION> ::= <RELATION> XOR <RELATION> S> :XOR12

114 | <XOR_EXPRESSION> XOR <RELATION> S> :XOR12

115 <ANDTHEN_EXPRESSION> ::= <RELATION> AND THEN <RELATION> S> :ANDTH12

116 | <ANDTHEN_EXPRESSION> AND THEN <RELATION> S> :ANDTH12

117 <ORELSE_EXPRESSION> ::= <RELATION> OR ELSE <RELATION> S> :OREL12

118 | <ORELSE_EXPRESSION> OR ELSE <RELATION> S> :OREL12
<SIMPLE_EXPRESSION> ::= <SIMPLE_EXPRESSION> <REL_OP> <SIMPLE_EXPRESSION> S> !112
REL_OP ::= = S> :I0
| /= S> :NI
| > S> :G1
| < S> :GE
| <= S> :EI
| <= S> :II

<SIMPLE_EXPRESSION> ::= <SUM>
<REL_OP> ::= <SIMPLE_EXPRESSION> <SIMPLE_EXPRESSION> S> <SIMPLE_EXPRESSION> S>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <UN_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <ADD_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <SUB_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <MUL_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <DIV_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <POW_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <MOD_OP> <_TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <REM_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <FLOOR_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <CEILING_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <TRUNCATE_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <ROUNDUP_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <ROUNDDOWN_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <SIGN_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <ABSOLUTE_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <NEGATIVE_OP> <_TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <POSITIVE_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <INFINITY_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <MINUS_INFINITY_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <MAXIMUM_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <MINIMUM_OP> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <PRODUCT_OPERATOR> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <DIVISION_OPERATOR> <TERM>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <EXPRESSION_LIST> S>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <EXPRESSION_LIST> S>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <FUNCTION_CALL> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <IF_EXPR> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <CASE_EXPR> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <TYPE_COERCION> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <TYPE_CASE_EXPR> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <INDEXOF_EXPR> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <INDEXOF:expr> <EXPR>
SIMPLE_EXPRESSION ::= <SIMPLE_EXPRESSION> <INDEXOF:EXPR> <EXPR>

<<SIMPLE_EXPRESSION> ::= <SIMPLE_EXPRESSION> <PLUS_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <MINUS_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <MUL_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <DIV_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <POW_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <MOD_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <REM_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <FLOOR_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <CEILING_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <TRUNCATE_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <ROUNDUP_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <ROUNDDOWN_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <SIGN_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <ABSOLUTE_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <NEGATIVE_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <POSITIVE_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <INFINITY_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <MINUS_INFINITY_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <MAXIMUM_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <MINIMUM_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <PRODUCT_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <DIVISION_OPERATOR> <TERM>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <EXPRESSION_LIST> S>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <FUNCTION_CALL> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <IF_EXPR> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <CASE_EXPR> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <TYPE_COERCION> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <TYPE_CASE_EXPR> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <INDEXOF_EXPR> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <INDEXOF:expr> <EXPR>
SB_ATOM > <TERM> ::= <SIMPLE_EXPRESSION> <INDEXOF:EXPR> <EXPR>
A V O C A B U L A R Y  C R O S S - R E F E R E N C I N G

& 162
( 33 34 43 44 45 54 103 140 146 152 153 154 155 156 157 158 159
) 33 34 43 44 45 54 103 140 146 152 153 154 155 156 157 158 159
* 135
** 1 1
+ 168
  36 38 39 40 41 42 49 68 170 172 173
  165 169
.etconst 72
.integer 137
.procname 8 9 48 49 71 150
.string 138
.typemark 73
.varname 27 37 38 39 40 67 70 95 100 101 149
/ 123
  27 28 57 67 100 101 160
  161
  67
  4 4 10 11 12 13 14 15 55 62 63 82
< 126
<= 127
= 122
>= 91 100 101
> 124
>= 125
AND 109 110 115 116
BEGIN 59 106
BODY 9
CASE 88 88
DECLARE 104
IF 87 117 118
IIF 86
END 8 9 59 87 88 97 106
FUNCTION 43 52
IF 84 87
INJ 163
IS 8 9 29 30 50 51 88 97
LENGTH 167
NOT 166
OR 111 112 117 118
OTHERS 96 102
PACKAGE 8 9
PRIVATE 31
PRO 164
PRODUCT 44
RETURN 58 83
SEQUENCE 46
SUM 45
SASI 97 97
THEN 84 86 115 116
TYPE 29 43 44 45 46
USE 47
WHEN 91 100 102
XOH 113 114
| 147 148
| 147 148
| | 92
< EXPRESSION LIST > 146 172 -172 -173
< LENGTH IDENTIFIER List > -41 42 -42 44 45
< LENGTH IDENTIFIER List > -28 -39 40 -40
< ADD OP > 133 -168 -169
< ADVANCED PROGRAM > 1 -2
< ALTERNATIVE > 89 90 -91
< ALTERNATIVE LIST > 88 89 -89 -90
< AND THEN EXPRESSION > 79 -115 116 -116
< AND EXPRESSION > 76 -109 110 -110
TOOLS FOR TESTING DENOTATIONAL SEMANTIC
REFERENCES


2. Stoughton, A., V. Kini, and D. Martin. AFDL+ Transcripts. Transcripts of Interlisp sessions during which the definition of the TINY language is processed through AFDL + tools.


