Generation of Ada¹ and PL/1 Prototypes

from Abstract Data Types Specifications

Boumediene Belkhouche

Computer Science Department
School of Engineering
Tulane University
New Orleans, LA 70118
bb@rex.cs.tulane.edu

Abstract

A software system prototype is an operational model that exhibits the behavioral and structural characteristics of the desired software product. We describe a prototyping system that automatically generates compilable prototypes by transforming an abstract data type specification into a program. The prototyping system consists of two versions: a compiler on MULTICS that generates PL/1 code, and a compiler on Unix² that generates Ada code. The proposed approach allows the specification developer to investigate the behavior of the specifications and to define implementation models.

Keywords and Phrases

Abstract data types, executable specifications, prototypes, prototyping system, implementation model, model fine-tuning, specification validation.

¹Ada is a trademark of the US DoD (AJPO).
²Unix is a trademark of AT&T.
1. Introduction

One of the objectives of requirements engineering is to express formally the somewhat fuzzy needs of the client. The formal expression constitutes a specification. The transition from requirements to specifications depends heavily on the intuitive and pragmatic capabilities of the software engineer to describe a software model that captures real-world objects. In situations where the real-world object does not exist already, drawing a picture or constructing a paper model may be used to visualize the object under study. The abstract nature of software does not yield to this type of concrete visualization. Consequently, software developers are faced with the task of building an actual running system that can be used to study the corresponding model. The cost-effectiveness and the timely availability of this system are critical to further developments. Software prototyping addresses the issues of cost, effort, availability, and specification validity [Squ82, Boe84, Bud84]. The objective of prototyping is to provide an initial implementation that is used to investigate the behavior of the system, to validate the specifications, to perform feasibility studies, and to define a base model for the final product implementation.

Existing approaches to prototyping emphasize the scaled-down nature of prototypes [Bud84, Tan89]. Their method can be two-fold. In the vertical strategy, a small set of critical functions is fully implemented and assessed. In the horizontal strategy, a skeleton of all the functions is implemented with an emphasis on the interfaces rather than the behaviors. Unlike these approaches, our method provides the tools to completely specify and to automatically generate software models. The major tools are a formal specification language for abstract data types (SUSL) and its processor [Bel86a,b]. In this case, prototyping is achieved through the transformation of a SUSL specification into its corresponding program. Thus, the prototyping strategy is both vertical and horizontal. The resulting prototype constitutes a preliminary, yet full-scale, implementation of the specifications. As such, a prototype is an
operational model that exhibits the behavioral as well as the structural characteristics of the desired software product. This definition of prototyping is consistent with the use of this word in other engineering disciplines (e.g., automotive, aerospace). Subsequently, the term prototyping is used in this context.

The proposed prototyping approach is discussed in the following sections. An overview of the methodology is given in Section 2. The actual implementation of the prototyping system is described in Section 3. Results collected from several experiments with the system are presented and discussed in Section 4. An example to illustrate the language and the prototyping environment is given in the appendix. The reader is referred to [Bel86a] which covers in detail the SUSL compilation process.

2. Software Development Methodology

Prototypes may be developed either manually or automatically. Manual prototyping involves hand-coding the prototype from the requirements [Boe84]. This is often referred to as a "quick and dirty" initial implementation. This activity is labor-intensive and may lack structure since it tends to bypass the specification and design stages. The invested effort and the pitfalls can be as significant as developing the real system. This fact may actually influence the developers to adopt the prototype as the final product. Another approach supports the automatic prototype definition from formal specifications by providing the necessary support tools to execute the specifications. Executability of specifications raises a number of issues among which are the degree of formality of the language, its operational semantics, and the efficiency of the tool and the product. At one end of the spectrum lie natural languages whose utility in prototyping has been very limited; and at the other end, we find formal specification languages that can be either compiled or interpreted, and that have shown great promise. The system discussed in this paper
supports the rapid generation of efficient prototypes by translating formal specifications into executable imperative languages (e.g., Ada, Pascal, PL/1). Specification-oriented prototyping can be situated at the low-level end of the spectrum.

The software development paradigm that our prototyping environment supports includes three major phases: requirements, specifications, and testing (Fig. 1).

2.1. Formal Specifications

The modelling technique is based on data abstraction. The abstract data type specification language SUSL [Bel86b] is used to formally specify the system described by the user’s requirements. An environment for compiling and executing specifications expressed in SUSL is implemented on two different operating systems [Bel86a]. The adoption of the data abstraction approach is motivated by the formal foundations, modularity, encapsulation, and parameterization of abstract data types. Moreover, it is our contention that abstract data types are best suited as reusable parts in a software factory environment, because of the nice attributes they possess. A formal definition of SUSL can be found in [Bel86b] and an overview is given in [Bel86a]. The formal definition consists of a BNF description of the grammar rules and a denotational semantics of the language.

The basic unit in SUSL is the abstract data type specification which is structurally similar to an Ada package. A specification provides two important features of a software module. The first feature is the interface which describes how a data type interacts with its environment. The interface consists of
import and export operations and types. The second feature is the semantics of the operations which define the functional behavior of the data type. The behavior is expressed in terms of input/output assertions. An assertion is a predicate on the observable state of the data type. An input (output) assertion captures the desirable initial (final) state of the data type. Operationally, an assertion can be viewed as describing the relationships among all the variables of the data type. Appendix A contains an illustrated description of a SUSL specification.

2.2. Testing of Executable Specifications

Testing is used to evaluate the behavioral characteristics of the prototyped system. Of importance are those characteristics that pertain to functionality and to performance. The role of functional testing is to ascertain that the implementation of a system is valid with respect to the corresponding specification [Mye79]. Generally, white box testing is performed, whereby criteria derived from the program structure are used to generate test cases. The main defect of this approach is that it exercises the functionality of the program independently of any specification, thus making the specification irrelevant. On the other hand, the availability of a compiler for SUSL effectively supports specifications testing. Test cases to excercise a specification are generated according to traditional methods used in program testing [Whi87]. This white box approach to testing the specifications is equivalent to black box testing of programs, except that it is performed very early in the software development process. Thus, errors due to the specification process are caught and repaired at this critical stage. As a result, the software product is more cost-effective and of a higher quality.

The expression of the functionality of each operation of an abstract data type specification in terms of input/output assertions lends itself naturally to boundary-value analysis. For example, consider the push operation of the stack specification shown in figure A.1 (Appendix A). Its specification is:
procedure push (inout s:stack; in elmt: T) signals (overflow)
begin
    given (n > #s)
    find s such that s = s' + elmt
end push;

From the input assertion "given (n > #s)", we identify three equivalence classes: {n: n > #s}, {n: n = #s}, and {n: n < #s}. We select the following set of test cases: {n = #s, n = #s + 1, n = #s - 1}, thus covering the boundaries of the input space. From the output assertion "s = s' + elmt", we identify two equivalence classes: {s: s = s' + elmt}, and {s: s ≠ s' + elmt}. We select the following set of test cases: {s = s' + elmt, s ≠ s' + elmt (i.e., overflow)}, thus covering the boundaries of the output space. In general, the assertions are relational expressions of the form:

expression₁ RELATIONAL_OPERATOR expression₂

Such expressions define three equivalences classes for testing purposes. These classes are characterized by: S₁ = {x: expression₁ = expression₂}; S₂ = {x: expression₁ ≠ expression₂}; and S₃ = {x: expression₁ > expression₂}, where x is the vector consisting of all the variables declared in the function being tested.

The role of performance testing is to assess the impact of various decisions concerning the data structure selection process (see section 3.2). A critical factor affecting the performance of a system is the data structures in the implementation. For instance, the implementation of a specification based on the abstract data structure set has various performance levels. The variations depend mainly on two parameters: the type of operations used in the specification and the concrete data structure selected to implement the set. The specification developer has control over these parameters, and thus, can affect the weights used by the SUSL processor in generating an implementation. As a result, different implementation models are defined and tested. An alternative is to fix one parameter and vary the other to effect model fine-tuning.
3. Overview of the Prototyping Environment

The prototyping environment consists of the SUSL specification language, and two different versions of the SUSL compiler. The first and initial version translates a specification into PL/1 code, and is available on MULTICS. The second version translates a specification into Ada code, and is available on a Pyramid 90x running Unix [Bel86a, Bel86b]. The components of the integrated environment are:

1. A specification language for abstract data types based on the abstract model specification technique [Hoa72, Lis75, Wulf76]. The language is based on a model that has been formalized, and shown to be powerful enough to express a large class of abstract data types [Ber79].

2. A prototype generator that transforms a specification of an abstract data type into an executable PL/I (or Ada) prototype program. This activity can be viewed as a VHLL compilation, where the source program is expressed in SUSL and the machine code is expressed in Ada or PL/1.

3. A preprocessor that integrates the prototype programs with other programs that use them. The preprocessor is necessary in the PL/1 environment, because PL/1 does not support abstract data types.

4. An abstract data type library manager that maintains information about specifications, reusable components, interfaces, instantiations, and implementations.

A brief scenario of the prototyping process is captured by the following procedure.

develop an initial version of the specification
compile the specification
test-run the specification
while not satisfied
    update the specification
    compile the specification
    test-run the specification
end while.
Thus, the main human task in the prototyping process involves the derivation of the formal specification of the system. The tools that support the prototype definition activity allow the software developer to gain insights into the behavior of the implemented model. As such, the paradigm prescribed by the prototyping system offers an alternative to the traditional software life cycle. It allows software engineers to concentrate all their efforts on refining specifications and assessing their validity through testing and experimentation.

3.1. Implementation

Figure 2 shows the logical organization of the implemented system.

![Software Prototyping Environment](image)

**Fig.2. Software Prototyping Environment**

The implemented system consists of the following phases:
The front-end consists of a parser and an attributes propagator. The parser generates an evaluation tree, a symbol table, and a type hierarchy table. The tree and the tables are input to the propagator to perform type-checking and overloading resolution. The result is a tree decorated with attributes. This tree, expressed in an abstract intermediate language (AIL), is passed on to the back-end. The back-end consists of a transformer (data structure and algorithm selection subsystem) and a code generator. These are discussed in the following sections.

3.2. Data Structure Selection

SUSL provides four abstract data structures: sets, sequences, direct products, and discriminated unions. Each abstract data structure has potentially several concrete representations in the target language. For example, sets may be implemented as arrays, linked lists, hashed tables, bit vectors, etc. The problem of selecting efficient representations encompasses a variety of factors, many of which cannot be measured by static analysis approaches. Not only does the selection process depend on partial information, but it is also routinely confronted with conflicting optimization criteria on a global level. The purpose of the data structure selection process is to map an abstract data structure into a concrete data structure given a set of selection criteria. Usually these criteria include space and time complexity measures that need to be minimized. That is, a concrete representation is selected only if its space and time costs are minimal with respect to other representations [Low76, Low78]. The parameters that influence the selection process are: (1) the size of the data structure; (2) the type of the operations performed on the data structure; and (3) the usage frequency of each operation. Parameters 1 and 3 are computed dynamically, whereas parameter 2 is computed statically.

The model adopted in this implementation is based on two sets of knowledge: a static body of knowledge derived from predefined mappings between abstract and concrete representations, and a dynamic body of knowledge derived from the specification being compiled.
3.2.1. Static Knowledge Derivation

Simulation was used to acquire the static knowledge. The set abstract data structure is used to illustrate the knowledge acquisition process. The simulation involved the following tasks:

1. Definition of a computational model to serve as a basis for complexity analysis. The RAM model [Aho74] was selected.
2. Selection of a kernel of operations associated with each abstract data structure. For sets, the kernel operations are add_an_element, and membership.
3. Selection of a set of concrete representations for each of the four abstract data structures. For sets, arrays, linked lists, and hashed tables form the set of concrete representations.
4. Implementation of the kernel operations for each concrete representation using this computational model. The RAM instruction set was used to implement the kernel operations using every concrete representation.
5. Space and time complexity analysis of the implementations. The space and time complexity measures of the implemented algorithms were analyzed.
6. Abstract implementation of non-kernel operations using the kernel operations. An abstract Algol-like language was used to express the semantics of non-kernel operations in terms of kernel operations. This strategy supports a representation-independent of the non-kernel operations, and facilitates the complexity analysis.
7. Space and time complexity measures of the non-kernel operations were computed in terms of kernel operations.

The computed complexity measures for all available representations for each of the abstract structures form the basis for the static knowledge. They are subsequently used as weight factors in the data structure selection process. Table 1 shows a partial list of set operations and their parameterized cost measures. Each concrete representation for add and member dictates a RAM implementation which, in turn, provides the numeric complexity measure. The parameter N represents the set size for unary operation and the size of the largest set for binary operations. The cost measure for empty and cardinality is 1, because both operations require only a reference to a set descriptor.

Obviously, the numeric complexities (A and M) of add and member depend on the particular concrete representation under consideration and on the model in which these two operations are expressed. For a hash table representation M is 1*H_1, whereas for an array representation M is N*H_2. H_1 and H_2 represent the complexity of the algorithms that implement the corresponding operations in a given model. Varying the basic model results in different values for the H’s. These variations affect the weight factors which, in turn, influence the selection process.

3.2.2. Dynamic Knowledge Derivation

The collection of information concerning the behavior of a specification is performed during the analysis phase of compilation. The frequency count of all the operations on abstract data structures is computed, and used in conjunction with the weight factors to build an ordered list of potential concrete representations. The following algorithm constructs the list:
-- Given a set of abstract data structures \{A, B, C, ...\}
-- Assume we are dealing with abstract data structure C.
-- Assume there are N operations and M representations for C.
-- Let CW \((i,j)\) be the weight factor for the implementation of operation \(i\) using representation \(j\). CW is precomputed.
-- Let CF \((i)\) be the frequency count for operation \(i\). CF is computed by the compiler.
-- Let CC \((i,j)\) be the cost of operation \(i\) using representation \(j\). CC is precomputed.
-- Let CT \((i)\) be the total cost for representation \(i\). CT is computed by the compiler.

\(L \leftarrow {}\) -- This is the ordered list of representations. It is initially empty.
CT \(\leftarrow 0\)
for \(i\) in \([1..N]\) loop
    for \(j\) in \([1..M]\) loop
        CC \((i,j)\) \(\leftarrow CF \((i)\) \ast CW \((i,j)\)\)
    end loop
    CT \((i)\) \(\leftarrow CT \((i)\) + CC \((i,j)\)\)
end loop
-- CT is sorted in non-decreasing order. As a result \(L\) contains the representations in non-decreasing order of cost.
-- \(L \left(1\)\) is selected as the optimal representation for C.
sort (CT, L)

The combined static and dynamic knowledge is used by the data structure selection system to propose an efficient implementation. The system has also the capability to interact with the software developer in cases where more information is needed. For example, the system is not able to derive future usage patterns of a given model, and thus, may ask the software developer to provide more information about such patterns.

### 3.3. Code Generation

The implicit nature of control structures in SUSL and its very high level operators and operands require several types of transformations to effect final code generation. The first stage in the transformation process involves the generation of explicit control structures and the expansion of the abstract operators. For instance, consider the set constructor \(\{x \in S: P(x)\}\) which defines the set of all \(x\)'s satisfying the predicate \(P\). This is transformed into:

\[
\text{temp_set} \leftarrow {}
\]
\text{enumerate (S, x)}
\text{\quad if P(x) then temp_set \leftarrow temp_set + x}
\text{\quad end if}
\text{end enumerate}
The second stage follows the data structure selection process. Once a representation is chosen, the system has enough information to generate the concrete algorithm corresponding to the abstract operations in terms of the selected representation. The result is expressed in a concrete intermediate language (CIL). Assuming that an array representation is chosen for the above example, then the CIL code is:

\[
\begin{align*}
  k & \leftarrow 0 \\
  j & \leftarrow \text{size (S)} \\
  & \text{for } i \leftarrow 1 \text{ to } j \\
  x & \leftarrow S[i] \\
  & \text{if } P(x) \\
  & \quad \text{then begin} \\
  & \quad \quad k \leftarrow k + 1 \\
  & \quad \quad \text{temp_set}[k] \leftarrow x \\
  & \quad \text{end} \\
  & \text{end if} \\
  & \text{end for}
\end{align*}
\]

The coder maps the constructs of CIL into constructs of the target programming language (Ada or PL/1 in this case). The inputs to this module are the type specification expressed in CIL, the set of data structures, and the symbol tables. The outputs are a file containing the executable code that will realize the type specification, and a file containing the interface declarations that will be included in the using programs. The coder module performs a direct mapping between the constructs of CIL and those of the target programming language. This mapping is very similar to source-to-source transformations. Were it necessary to generate code in another block-structured language (e.g., C or Pascal), it would suffice to redefine the syntactic mapping between the source and target language constructs. A detailed description of the code generation process is described in [Bel86a].

### 3.4. The Integrated System

An abstract data type is intended to be used by other Ada or PL/1 programs. The compiler generates the source codes implementing the specifications, and does not interact with any other subsystem. In the case of Ada, the `use` clause is used to achieve the interaction between using programs...
and programs implementing the specifications. For PL/1, an interface system to facilitate the interaction was also implemented. The function of this system is three-fold:

1. It processes all the abstract data type declarations in the using programs and generates the proper PL/1 declarations.
2. It incorporates the interface declarations within the using programs.
3. It processes all the abstract data type operations within the using programs and generates the proper PL/1 call statements.

In addition, the system interacts with the abstract data type library manager, where information about specifications, instantiations and implementations of abstract data types is maintained. The library manager was formally specified using SUSL. Information in the library is available to the compiler for checking interfaces, and to users for locating specific components. A heuristic strategy based on a semantic classification of software components is being implemented. The strategy allows software developers to query the library for reusable abstract data types.

4. Evaluation Results

Substantial gains in software development can be achieved through rapid prototyping. The integrated system supports this fact by demonstrating the feasibility of executing, testing and fine-tuning specifications. However, a complete assessment of the advantages and disadvantages of the proposed approach with respect to the traditional software life cycle approach requires answers to several questions, among which:

1. How efficient are the prototypes with respect to hand-coded systems?
2. How many errors are detected through prototyping that would have otherwise gone undetected?
3. How fast and how easy is the prototyping process with respect to the traditional software life cycle process?
4. How does prototyping costs compare with traditional software life cycle costs?
Because the construction of specifications is a common factor to all software development paradigms, a comparison of speed and efficiency of the prototyping and the traditional processes reveals the benefits of prototyping. Speed of development is not an issue in prototyping, since compiling and executing a specification takes orders of magnitude less than hand-coding programs. Consequently, code size and run time become the determining factors in the comparison.

Three small experiments were carried out to assess the quality of the prototypes generated by the system, and to show that prototypes compare favorably with hand-coded programs. The first experiment consisted of specifying five abstract data types, and then comparing hand-coded and system-generated programs implementing the specifications. The size in MULTICS words of the source code of the respective programs is given in Table 2. The ratio of the average of column 2 over column 3 is 1.42. This means that in this sample, the synthesized programs are 42% larger than the hand-coded ones.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Synthesized Program Size</th>
<th>Hand Coded Program Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque</td>
<td>1034</td>
<td>695</td>
</tr>
<tr>
<td>list</td>
<td>1304</td>
<td>3002</td>
</tr>
<tr>
<td>set</td>
<td>3029</td>
<td>3098</td>
</tr>
<tr>
<td>stack</td>
<td>926</td>
<td>403</td>
</tr>
<tr>
<td>symtab</td>
<td>7643</td>
<td>2591</td>
</tr>
<tr>
<td>average</td>
<td>2788</td>
<td>1958</td>
</tr>
</tbody>
</table>

Table 2: PL/1 Source Code Sizes

The same data were collected for the Ada experiment. In this case, the size is expressed as a word count (what is returned by the function wc on Unix). Table 3 shows the sizes. The ratio of column 2 over column 3 is 1.45. This means that the synthesized programs are 45% larger than the hand-coded ones. The small difference between the two ratios (0.42 and 0.45) seems to be a coincidence.
Table 3: Ada Source Code Sizes

The second experiment dealt with PL/1 object codes for hand-coded and synthesized programs. Table 4 shows the sizes. The ratio of column 2 over column 3 is 1.32, which is quite close to the ratios for sources codes.

Table 4: PL/1 Object Code Size Comparison
The size measurements do not provide a complete indication of relative efficiency. An appropriate method to measure efficiency is to conduct a performance analysis of the programs by executing them. This task was performed through the third experiment. A version of the shunting algorithm for arithmetic expressions was used as a driver and the stack program as the abstract data type. Six expressions were translated by the algorithm into postfix notation. Each expression was processed 100 times, and the average time in microseconds recorded. The results of the experiment are shown in Table 5. The ratio of the averages of column 2 over column 3 is 1.21. This means that the synthesized programs were 21% slower than the hand-coded ones.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Synthesized Program Time</th>
<th>Hand Coded Program Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c</td>
<td>18248</td>
<td>16773</td>
</tr>
<tr>
<td>a<em>b</em>c</td>
<td>23819</td>
<td>15229</td>
</tr>
<tr>
<td>a+b*c</td>
<td>17225</td>
<td>21393</td>
</tr>
<tr>
<td>a*b+c</td>
<td>18635</td>
<td>15126</td>
</tr>
<tr>
<td>a+b*c+d</td>
<td>32446</td>
<td>21571</td>
</tr>
<tr>
<td>a<em>b+c</em>d</td>
<td>26803</td>
<td>22357</td>
</tr>
<tr>
<td>average</td>
<td>22803</td>
<td>18742</td>
</tr>
</tbody>
</table>

Table 5: Run Time Comparisons for PL/1 Programs

6. Conclusion

Given that software prototyping is considered as an initial development effort, the data shown in this paper suggest that prototyping through executable specifications can be relatively efficient. The major effort in the proposed software development paradigm is in the generation of the specifications in SUSL. However, this effort is necessary no matter which paradigm is adopted. Consequently, the prototyping environment contributes significantly to the cost-effectiveness of developing software. The usefulness of the prototypes is not limited to specification testing. They are efficient enough to be used
effectively throughout the software development stages. These prototypes can also serve as template models for the production modules. Not only does the proposed approach provide a tool, but it defines a methodology for software development. The stages of the new software cycle are more reliable and cost-effective. The objectives of this research were: (1) to provide a credible alternative to the waterfall model of software development; (2) to demonstrate the feasibility of compiling the SUSL specification language; and (3) to provide support for specification testing. The implemented system and the results presented here support these objectives.

Acknowledgements. I would like to thank all the graduate students who were involved with the SUSL project. Peggy Jordan, Lisa Levy and Marguerite Saacks evaluated and redesigned the language. Michael Bringmann implemented the new version on the Pyramid. Bart Geraci developed the formal definition of the language. I also would like to thank Johnette Hassell for her support.
References


[Squ82] Squires, S., Special Issue on Rapid Prototyping, ACM Software Engineering Notes 7,5 (December 1982).


Appendix A

The example shown in Figure A.1 will be used to give an overview and to illustrate the use of the prototyping system. The
input to the prototyping system is a file containing the abstract data type specification. The output is a compilable program. The
PL/1 version is shown in Figure A.4 and A.5, and the Ada version is shown in Figure A.6.

1. type stack (T: type; n: integer) given n > 0;
2. interface
3. export operations:
4. create returns (stack);
5. push (inout stack; in T) signals (overflow);
6. pop (inout stack) signals (underflow);
7. top (stack) returns (T) signals (no_top);
8. empty_ (stack) returns (boolean);
9. depth (stack) returns (integer);
10. types stack = sequence (x: T, n);
11. initially stack = {stack:};
12. operations
13. function create returns (s:stack)
14. begin
15. find s such that s = {stack:}
16. end create;
17. procedure push (inout s:stack; in elmt: T) signals (overflow)
18. begin
19. given (n > #s)
20. find s such that s = s’ + elmt
21. end push;
22. procedure pop (inout s: stack) signals (underflow)
23. begin
24. given (0 < #s)
25. find s such that s = head (s’)
26. end pop;
27. function top (s: stack) returns (elmt: T) signals (no_top)
28. begin
29. given (0 < #s)
30. find elmt such that elmt = last (s)
31. end top;
32. function empty_ (s: stack) returns (b: boolean)
33. begin
34. find b such that b = empty (s)
35. end empty_;
36. function depth (s: stack) returns (d: integer)
37. begin
38. find d such that d = #s
39. end depth;
40. end stack.

Fig. A.1. SUSL Specification of The Type Stack

Note that line numbers in Figure A.1 are used to cue references in the discussion below. Each SUSL specification unit
consists of the following sections:
(1) a type header describing the type name, optional generic parameters, and constraints on the parameters (line 1);
(2) an interface describing the imports types, the export types, and the export operations (lines 2-9);
(3) a type definition describing the structure and the defining types used to model the principal type (line 10);
(4) an initialization describing the initial value of a variable bound to this type (line 11);
(5) a body describing the semantics for each operation of the type (lines 12-39);
(5a) operations may be procedures (lines 17, 22) or functions (lines 13, 27, 31, 36);
(5b) all operations may define input assertions (lines 19, 24, 29) which raise exceptions in case they are not satisfied;
(5c) all output assertions are computed by functional expressions (lines 15, 20, 25, 30, 34, 38)
(6) a tail bracket used to signify the end of the unit (line 40).
It should be noted that the specification in Figure A.1 defines a new type (stack) that can now be used by other specifications or programs. This new type is treated the same way primitive types are.

A program using an abstract data type must have a use section for visibility, a type section for instantiations, and a var section for declarations to be able to use an abstract data type. The processing of these sections is performed in order to generate declarations for abstract data type variables, to generate initialization calls, and to build an abstract data type instantiation file. This instantiation file is used by the prototyping system to generate different implementations for different instantiations.

Figure A.2 shows an example of a using program, and Figure A.3 shows its abstract data type declaration file.

```pl1
stack_driver: proc;

/* statements in brackets are non PL/1 */
/* They are translated into PL/1 code by the system */
[stack_decls]

dcl e fixed bin;
dcl len fixed bin;
dcl state bit (1);

s = [int_stack$create];
r = [real_stack$create];
e = 10;
call [int_stack$push (s,e)];
len = [int_stack$depth (s)];
state = [int_stack$empty_ (s)];
e = 11;
call int_stack$push (s,e);
e = 0;
e = [int_stack$top (s)];
call int_stack$pop (s);
end /* stack_driver */;
```

Fig. A.2. Example of a Using Program

```pl1
use /* visibility */
stack (T: type; n: integer);

type /* instantiations */
int_stack = stack (integer;100);
real_stack = stack (real;20);

var /* declarations */
s : int_stack;
r : real_stack;
```

Fig. A.3. Abstract Data Type Declaration File

The PL/1 prototyping system generates two files: a code file shown in Figure A.4, and an interface declaration file shown in Figure A.5. The Ada prototyping system generates a single file that includes the package specification and body. This file is shown in Figure A.6.
stack: proc (n, stack__00);
    dcl stack__00               ptr;
    dcl t__01                   ptr;
    dcl s                       ptr;
    dcl t__03                   fixed bin;
    dcl t__04                   bit (1);
    dcl t__05                   fixed bin;
    dcl n                       fixed bin;
    dcl t__07                   bit (1);
    dcl t__08                   bit (1);
    dcl elmt                    T;
    dcl t__010                  ptr;
    dcl t__011                  T;
    dcl t__012                  T;
    dcl b                       bit (1);
    dcl l                       fixed bin;
%include rts_decls;
stack__00 = seq$empty ();
return;
create: entry (n) returns (ptr);
s = seq$empty ();
return (s);
push: entry (n,s,elmt);
dcl overflow                condition;
call rep$sequence size (t__03,s);
t__04 = 0 <= t__03;
call rep$sequence_size (t__05,s);
t__07 = t__05 < n;
t__08 = t__04 & t__07;
if   t__08
  then do;
    dcl temp__01                   ptr;
    call rep$copy_sequence (t__01,s);
    allocate T set (temp__01);
    temp__01 -> T = elmt;
    call rep$radd sequence_element (t__01,temp__01);
    end;
s = t__01;
  end;
else signal overflow;
return /* push */;
/* bodies for the other operations go here */
end /* stack */;

Fig. A.4. PL/1 Program Prototype For The Specification of Figure A.1

dcl stack        entry (fixed bin,ptr);
dcl stack$create entry (fixed bin) returns (ptr);
dcl stack$push   entry (fixed bin,ptr,T);
dcl stack$pop    entry (fixed bin,ptr);
dcl stack$top    entry (fixed bin,ptr);
dcl stack$empty_ entry (fixed bin,ptr) returns (bit (1));
dcl stack$depth  entry (fixed bin,ptr) returns (fixed bin);

Fig. A.5. Interface Declaration File
generic
type T is private;
n : integer;
package stack is
  type stack is limited private;
  function create return stack;
  procedure push (P1 : inout stack; P2 : in T);
  procedure pop (P1 : inout stack);
  function top (P1 : stack) return T;
  function empty_ (P1 : stack) return boolean;
  function depth (P1 : stack) return integer;
private
  type ztrstack;
  type ztastack is access ztrstack;
  type ztrstack is
    record
      item : T;
      next : ztastack;
    end record;
  type stack is
    record
      maxsize : integer := n;
      cntsize : integer := 0;
      thisndx : integer := 0;
      first : ztastack;
      last : ztastack;
      this : ztastack;
    end record;
end stack;
end stack;

-- Bodies of other functions and procedures deleted.

Fig. A.6. Ada Program Prototype For The Specification of Figure A.1.