Ref : 2010/Master MOCS/SEM Submitted on 10/08/2010

Conservatoire National Des Arts Et Metiers

Paris

A Graduation Project Report
Presented for obtaining

Master 2 degree

By

Marc SANGO

TITLE

Design and Implementation of a Verification Profile for Ada

Organism: AdaCore SAS
Supervised by : Yannick MOY
E-mail : yannick.moy@adacore.com
Cnam Supervisors : M.V. APONTE GARCIA & P. COURTIEU
Chairman of Board: Samia BOUZEFRANE
Jury : E. GRESSIER, J.F. SUSINI, M.V. APONTE, P. COURTIEU

RÉSUMÉ

La complexité des systèmes logiciels ne cesse d'augmenter, ce qui rend leur vérification difficile. Le coût et les garanties de cette vérification dépendent de façon essentielle du langage de programmation utilisé. Ada est un language de programmation largement utilisé pour la programmation d'applications critiques car orienté vers la sûreté et la sécurité des programmes, notamment grâce à un système de typage fort très riche (sous-typage, héritage de classes, portée des types pointeurs, etc.) vérifié statiquement par le compilateur. Le langage SPARK, basé sur Ada, permet d'aller plus loin dans la vérification formelle de propriétés du programme, en exprimant les contrats des sous-programmes par des propriétés énoncées en logique du premier ordre, à condition d'accepter des restrictions fortes sur les programmes (pas de pointeurs, pas d'aliasing, fonctions pures, etc.). Le projet Hi-Lite a pour ambition d'appliquer les capacités de vérification formelle de SPARK à des programmes beaucoup moins contraints, en se reposant sur la capacité de vérification dynamique (exécution de tests) quand la vérification statique n'est pas possible. Une des premières étapes de ce projet consiste à définir un sous-langage d'Ada aussi large que possible adapté à la vérification formelle par traduction automatique vers SPARK, que nous appellerons un profil de vérification pour Ada. Le but de ce projet de stage de master consiste à définir ce profil.

Mots-clés : Ada, SPARK, vérification, preuve de propriétés, test, analyse statique, méthodes formelles.

ABSTRACT

The complexity of software-intensive systems keeps growing, which makes program verification difficult. Cost and guarantees offered by program verification depend critically on the programming language used. Ada is a programming language largely used in critical software development because it is strongly oriented towards safety and security of programs. Its key feature is a very rich strong type-system (subtypes, class inheritance, scoping of pointer types, etc.) which is verified statically by the compiler. The SPARK programming language is based on Ada and allows further proofs of properties on programs, through the use of first-order logic contracts on subprograms, at the cost of strong restrictions on program features (no pointers, no aliasing, pure functions only, etc.). Hi-Lite project aims at applying formal verification techniques from SPARK to programs outside the SPARK boundaries, by relying on dynamic verification (testing) when static verification is not possible. The first step consists in defining a verification profile for Ada: that is, a larger subset of Ada suitable for formal verification by automatic translation to SPARK. The goal of this MSc internship is to define such a profile.

Keywords : Ada, SPARK, verification, proof of properties, testing, static analysis, formal methods.

Acknowledgment

My first thanks to my supervisor Yannick MOY, AdaCore Senior Engineer; thank you for welcoming me in the Hi-Lite project, for giving me the opportunity to develop this experience and for challenging me every day to make me a better engineer.
Also, special thanks to all people at AdaCore, you were always there to listen to me and give me some advice; in particularly, thank you Jérôme GUITTON for your continuous support.
My advisors at CNAM, M.V. APONTE GARCIA & P. COURTIEU are most thanked for helping me with their encouragement and their constant guidance during my internship. Thank you M.V. APONTE, for giving me the opportunity to join the Hi-Lite project.
Thanks also to others trainees and my classmates; I am also greatly indebted to all my teachers, Samia BOUZEFRANE, the master SEM responsible, and her teaching team.

À tous ceux qui se sentiront honorés par les travaux de ce projet.

M. SANGO

Glossary

Ada : The Ada Programming Language.
AST : Abstract Syntax Tree is a tree representation of the abstract syntactic structure of source code written in a programming language.
BNF : Backus–Naur Form is meta-syntax used to express a context free grammar to define the syntax of a programming language by using two sets of rules (lexical and syntactic).
Formal methods : mathematically-based techniques for the specification, development and verification of software and hardware systems.
LRM : Language Reference Manual.
Property : Functional description of the behavior of a system.
Ravenscar : The Ravenscar profile is a subset of the Ada tasking features designed for safety-critical hard real-time systems.
SPARK : SPARK is an annotated sub language of Ada, intended for high-integrity programming.
Static analysis : Static code analysis is the analysis of software that is performed without actually executing programs.
Testing : Investigation of a software system to detect bugs or verify safety and security properties, by means of selected executions.
VC : A Verification Condition is a mathematical formula, usually generated automatically, which must hold for some desired property of the system to hold.
Verification : Act of proving or disproving the correctness of program specification.

Table des matières

Liste des tableaux

General Introduction

Ada language and its toolkit are regarded by many companies as an ideal language for development of critical applications. Ada programming language is particulary well suited for static and dynamic verification in long-lived applications where safety, security, and reliability are critical.

Software becomes more and more complex and this affects all aspects of life when security matters. To prevent sofware failures or vulnerabilities to cause damage on their running environment, it is important that high-integrity software be thoroughly tested, verified, and certified. This ensures that safety and security are preserved during software's life-cycle. There are many levels where these activities can take place; among them the programming level that strikes a good balance between abstraction and details. The most common techniques used at this level are testing, model checking, static analysis and formal proof of properties.

The Hi-Lite project [1] aims at integrating these different verification techniques for a larger subset of Ada and C programming language. The project is lead by a consortium made up of two research institutes and five industrial partners, having AdaCore as the project leader. One project's goal is to provide similar tools to those already existing for Ada-Subset SPARK. SPARK is an annotated sublanguage of Ada, intended for use in safety-critical applications; its framework is based on powerful tools for automatically verifying many program properties. However, some users of Ada who try SPARK may be dismayed by the many restrictions of SPARK compared to Ada, hence the project to provide similar tools for a larger subset of Ada.

With the intent to automatically translate Ada programs into SPARK programs for verification (and not for execution), it was decided to define an appropriate *verification Profile for Ada* namely, a larger subset of Ada than SPARK, suitable for automatic translation into SPARK, and thereby still suitable for SPARK verification tools usage. Given an Ada program respecting all the *restrictions* in this profile, it would be possible to automatically translate it to SPARK so that existing SPARK tools could be used. The definition of such a profile is the objective of this internship. On one hand, we are interested in precisely identifying those Ada restrictions that really compromise automatic verification. On the other hand, we are interested in defining a set of SPARK extensions, that is, a set of Ada constructions lying outside the SPARK language, but suitable for automatic translation into it. In practice we will obtain a subset of Ada larger than SPARK to which SPARK tools can be applied for verification purposes.

Chapter 1 introduces AdaCore's activities as well as Hi-Lite project. Chapter 2 outlines our verification profile for Ada proposal, which includes suggestions as to how to translate from programs in this profile to SPARK. Chapter 3 details the technical design and implementation of a few proposed extensions with respect to SPARK. The detailed comparison between Ada BNF and SPARK BNF, based on their respective Reference Manuals, is given in Annex A. Chapter 4 describes what we have achieved during this internship, the difficulties we faced and the additional personal benefits we obtained.

1
Internship Context

1.1 Introduction

This MSc graduation project took place at AdaCore Paris society, in the context of Hi-Lite project lead by AdaCore. In the following, we describe shortly both AdaCore and Hi-Lite.

1.2 Company Presentation

Founded in 1994, AdaCore is the leading provider of commercial software solutions for Ada, a state-of-the-art programming language designed for large, long-lived applications where safety, security, and reliability are critical. The company is spread over America (AdaCore US) and Europe (AdaCore EU), based on an original cooperation between two legally distinct entities working as one.
AdaCore's flagship product is GNAT Pro, an open-source development environment with expert on-line support, available on many platforms. AdaCore's team also develops various tools, like the CodePeer tool for static analysis.

1.3 Hi-Lite Project Overview

1.3.1 Hi-Lite Project

The overall aim of Hi-Lite is to promote formal methods during the development of critical industrial applications. The idea is to allow developers of such applications not only to completely verify the safety properties and the logical properties of their software, but also to combine these methods with techniques for testing and static analysis, with a complete integration into their familiar development environment.

1.3.2 Objectives of the Internship

In the context of Hi-Lite project, the main goal of the internship is to define a larger subset of the Ada programming language that relaxes some of the restrictions in the SPARK subset of Ada. Indeed, SPARK is a programming language included in Ada, and equipped both with a specification language allowing the expression of desired properties for programs and with powerful tools for automatically verifying most of these properties. Ada programming language is heavily used for the development of critical applications; partly thanks to its intrinsic specification features, but there are no verification tools for Ada that compare to those existing for SPARK. In order to increase SPARK expressiveness an important step in this project is to identify a subset of Ada (larger than SPARK) that can be converted into SPARK. This conversion will allow to prove properties of Ada programs that do not respect some restrictions in SPARK with respect to Ada..

1.4 Internship Project Overview

1.4.1 Motivation

The Hi-Lite project motivation is to apply formal verification techniques to Ada programs (eventually C programs) by relying on dynamic verification (testing) when static verification is not possible. In Ada programming part of the project, the first step consists in defining a verification profile for Ada. Therefore, the motivation of the internship project consists in defining a larger subset of Ada suitable for formal verification by automatic translation to SPARK.

Our approach is to describe the profile and to test it in an Ada to SPARK prototype translator, described in section 1.4.3. The following section presents briefly the results we obtained using this approach.

1.4.2 Results of our Approach

Our approach in achieving the objective of the internship presented in section 1.3.2 is to describe a larger subset of Ada starting from SPARK while making sure that this subset really corresponds to a verification profile for Ada. In table 1.2 we present some statistics on our realizations of our work described later in this report.

Realizations	**statistics**
Suggested verification profile for Ada (see 2.3.2)	1
Restrictions proposed in the Profile (see 2.3.4)	13
Extensions proposed in the Profile (see 2.3.3.2)	26
Extensions realized in Sparkify (see 3.3)	4
Tests and demo added in Sparkify tests case (see 3.3)	20
Extensions proposed in SPARK Roadmap (see 2.3.3.1)	5
Ada and SPARK BNF comparisons (see A)	1
Minor inaccuracies in the documented SPARK BNF (see 2.3.1)	8

TAB. 1.2 – Results of our approach

The next section motivates the choice of technology used to test and debug the proposed Profile.

1.4.3 Choice of Technology

Many possibilities are available to convert to SPARK the subset of Ada defined in this internship:

▷ Python API of the GNAT front-end: It is the simplest solution to get a handle on the Ada AST for simple queries.

▷ ASIS-for-GNAT: This infrastructure defines a complete API to access (in read-only mode) the Ada AST, which conforms to the standardized Ada Semantic Interface Specification.

▷ A new back-end of the GNAT compiler: This would provide complete read-and-write access to the Ada AST.

A new back-end is the potential final solution. In fact, the major drawback of ASIS-for-GNAT is that its API only allows manipulating a read-only abstract syntax tree, while we are interested in modifying the syntax tree itself in order to generate SPARK from Ada. Both languages are indeed very similar, hence the idea of plugging the SPARK generator in the GNAT back-end, where this tree can be modified. This approach would have several other advantages: some operations that are needed by any existing back-end of the compiler are also useful when generating SPARK code. An example is the generation of named subtypes in place of anonymous ones.

In this internship, we adopted the solution based on ASIS-for-GNAT in order to develop quickly a prototype which purpose is to allowing us testing and debugging the solutions that we propose.

1.4.4 Sparkify Tool

Sparkify is an already Ada to SPARK translator based on the GNAT compiler and ASIS-for-GNAT. It takes a compilable Ada source as input and generates the transformed version of the source as output. Sparkify is an ASIS-based tool, which means that it accesses the AST from the compiler through the standardized Ada Semantic Interface Specification (ASIS). The ASIS implementation for GNAT, called ASIS-for-GNAT, is a separate product which is not included in the standard GNAT distribution.
The final goal for this tool is to translate Ada programs complying to a certain set of restrictions (a verification profile) into SPARK, and reject other programs. So, our objective is to add on this tool some call backs in order to testing and debugging the design of some extensions proposed.

In the following section we give the overview of Sparkify implementation.

1.4.5 Sparkify translating mechanism

The general architecture of Sparkify consists in various depth-first traversals of the AST. During the translation from Ada source to SPARK source, we maintain a cursor pointing to the first point in the source that was not treated previously (called Echo_Cursor).
When reaching an element where a special treatment is needed:

- we echo all the text from the source file on the output, from cursor Echo_Cursor to the current position;
- we do some special printing on the output;
- we may set cursor Echo_Cursor to skip some text in the source file when doing the next echo.

This can be done either during the pre-operations, when a node is treated before its children nodes are visited, or during the post-operations, when a node is treated after its children nodes have been visited.

In order to allow SPARK tools reporting issues back to the original source, correspondence to the original source file is maintained by printing special comments showing line directives, similar to what is done during C preprocessing.

1.5 Conclusion

In this chapter, we have given an overview of the company, the project and the internship, which includes a description of the Sparkify tool that we extended. The following step consists in identifying an Ada subset ideally suited for automatic verification. So, we will focus next on the major outcome of the report: specification and global design of a verification profile for Ada.

2
Specification and Global Design

2.1 Introduction

In this chapter, we define a verification profile for Ada. We define a *verification profile for Ada* as a larger subset of Ada suitable for formal verification by automatic translation to SPARK. We start in section 2.2 by looking at SPARK's high-level goals and Hi-Lite's objectives to motivate the need for the various *extensions* we propose to include in the Profile and for various limitations of SPARK we propose to keep in the Profile as a Profile *restrictions*. An *extension* means here an automatic translation from Ada to SPARK. The real extensions of SPARK language are expected in SPARK roadmap. In section 2.3.1, we define the proposed Profile by comparing step by step both Ada language [3] and SPARK language [4]. The BNF comparison coming from both LRM languages is summarized in Annex A. In section 2.4, we present some customer requirements that helped prioritizing the different extensions by translation that we implemented in the prototype tool to provide an operational prototype.

2.2 Comparison of SPARK and Hi-Lite Objectives

Ada is a high level language providing several desirable features (strong typing, modularity mechanisms, parallel processing, dynamic dispatch, exception handling, generics, ...). SPARK is an annotated subset of Ada, intended for high-integrity programming. Note that SPARK is not defined formally but its emphasis on a simple definition already makes the language and static analysis tools relatively robust. For historical reasons, SPARK lacks many Ada features that could be included without impairing the overall goal of SPARK providing a language and tools for making easier automatic formal verification. Some features initially excluded from SPARK (like Ada tasks) are now allowed (through the Ravenscar Profile which provides a method for constructing concurrent Ada programs with deterministic behavior) or have been added to the SPARK roadmap, an overall "to do" list for SPARK Pro [2] (This road map includes features based on SPARK customers requirements, like dynamic dispatching, generics and others).

In the context of Hi-Lite, our goal is to extend SPARK; i.e. define an unambiguous larger subset of Ada. Thus, we comment below in table 2.2 the comparison between SPARK and Hi-Lite to determine how SPARK and Hi-Lite work together..

Considerations	**SPARK**	**Hi-Lite**
1 : Unambiguous subset of Ada	✔	✔
2 : Enforce style	✔	✗
3 : Executable annotations	✗	✔
4 : Logical annotations	✔	✗
5 : Check data initialization	✔	✔
6 : Check data flow	✔	✗
7 : Check information flow	✔	✗
8 : Automatic proof	✔	✔
9 : Manual proof	✔	✗
10 : Combine test and proof	✗	✔

TAB. 2.2 – SPARK and Hi-Lite Comparison

1: SPARK has a different BNF from Ada and language rules restricting rigorously what is a valid SPARK program. Hi-Lite aims at defining a set of simpler restrictions with respect to Ada.

2: SPARK enforces its style rigorously not just to make the language strict but also to make tools for statics analysis of SPARK relatively simple and robust, while Hi-Lite project aims at minimizing the burden on programmers by not imposing a strict style of programming on them. In addition, the project is based on the existing GNAT compiler which already provides tools for coding style checking.

3: Making the annotations executable is one of the main objectives of Hi-Lite project. SPARK offers annotations but they are not executable. The ability to check properties at run time is both:
- a valuable default when proof does not succeed;
- a way to debug specifications.

4: Logical annotations are outside of the scope of Hi-Lite because they cannot always executed. This results in some loss expressiveness in specifications compared to SPARK (e.g. no unbounded quantification).

5, 6, 7: SPARK Examiner allows programs to be checked at several levels: data initialization, data flow, information flow. In the context of Hi-lite, checking data initialization is an objective, because it is possible to do it without requiring users to add specific annotations (provided the whole program can be analyzed), while the others are outside the scope of the project, because they would require additional work from users.

8: SPARK has two important tools: the Examiner that is able to generate verification conditions (VCs) that are processed automatically by the Simplifier, an automatic prover. One of the goals of Hi-Lite is to integrate additional automatic provers to discharge the VCs.

9: In order to complete a proof, SPARK allows the user to prove VCs manually with the assistance of a tool (Proof Checker) to check the validity of a proof script. In Hi-Lite, the focus is rather on adding other automatic provers.

10: Combining tests, static analysis and proof is the main challenge of Hi-Lite project; it bridges the gap between static and dynamic checking.

After this overview of SPARK and Hi-Lite goals we can begin, in the following section, to define the Profile.

2.3 Verification Profile Considerations

2.3.1 Comparison of Ada and SPARK BNFs

Here we compare Ada syntax and SPARK syntax and theirs semantic differences in order to determine the ideal profile for verification purpose. In this document we will call this subset of Ada the Verification Profile for Ada, or simply the Profile.

Many productions are exactly the same in Ada and SPARK. For other productions, SPARK LRM uses an asterix (*) before the production rule to indicate an existing non-terminal with different production rules with respect to Ada; and a plus (+) for new non-terminals. The full overview of our comparison between both BNFs is summarized in Annex A. During our comparison of Ada and SPARK BNFs, we detected some minor inaccuracies in the documented SPARK BNF which are listed below (reported in TN [1] J902-015).

1. Lack of an asterix (*) before a BNF 3.2.2: subtype_indication of SPARK differs from that of Ada since Ada 2005 includes null_exclusion in subtype_indication. (TN J610-023)
2. Forget to remove a (*) before a BNF 3.2.2: composite_constraint is now identical in Ada and SPARK, since discriminant_constraint was added for RavenSPARK. (TN J603-016)
3. In a rule of BNF 3.11, generic_function_instatiation (should be "instantiation") was introduced to allow instantiations of unchecked conversions some time ago but instead generic_instantiation (see BNF 12.3) was defined. (TN J614-007)
4. Minor inaccuracy in SPARK BNF summary syntax 5.1: statement is prefixed with the altered sign (*) in SPARK BNF while the two rules are identical in Ada and SPARK.
5. There should be no (*) before 5.1: <<label_statement_identifier>> production because both Ada and SPARK syntax is identical.
6. Minor inaccuracy in SPARK syntax rule 6.1: procedure_specification where (+) should be (*). Also, SPARK BNF prefixes the rule 6.1: function_specification by (+) while both rules in Ada and SPARK are the same. (TN J621-025)
7. There should be a (*) before BNF 6.1: parameter_and_result_profile.
8. There should not be a (*) before BNF 13.5.1: record_representation_clause of SPARK because it is identical to that of Ada.

Our ultimate concern is to define a verification profile for Ada. Since SPARK is a subset of Ada which provides some good characteristics for verification, we try to extend SPARK by translation in order to

[1] AdaCore work flow Ticket Number

provide a simpler and larger subset of Ada than SPARK. Thus we would be able to prove properties of Ada programs that do not respect some strict restrictions of SPARK.

In the following, we discuss rules that differ between Ada language and its subset SPARK language, in order to define the Verification Profile for Ada.

2.3.2 Description of a Verification Profile for Ada

There are two syntax descriptions in SPARK LRM: the language syntax of the core SPARK language as a subset of Ada and the syntax of proof contexts which are SPARK-specific. This specific annotation language allows to express and verify many properties on SPARK programs. Other languages define similar annotations for C (ACSL), Java (JML) and C# (Spec#). Our description of the Profile includes only the core syntax between Ada and SPARK.

Thus, in our Profile :

▷ There are no SPARK proof syntax, i.e. no SPARK annotations. We intend to generate two kinds of SPARK annotations from the Ada source code:

   1. data-flow contracts: These SPARK annotations specify which global variables are read and/or written by a subprogram. They are a prerequisite for using the SPARK toolset. We have designed and implemented a data-flow global analysis which computes these sets of variables for each subprogram in some complete Ada program.
   2. functional contracts: These SPARK annotations give constraints on the calling contexts of subprograms (precondition) and on the state at subprogram return (postcondition). We translate GNAT special pragmas Precondition and Postcondition into the equivalent SPARK ones. Likewise, we translate pragmas Assert into SPARK assert and check annotations. The next version of the Ada standard, expected to be initialized in 2012 contains many extensions that facilitate the expression of specifications as subprogram contracts or type invariants that, therefore, would be generated to corresponding SPARK annotations.

▷ We lifted certain restrictions from SPARK having no impact on verification tractability, as those related visibility, overloading, compile time constraints, renaming, and recursion.

▷ We kept the SPARK restrictions for the language constructions really compromising formal verification. So in the Profile there are no pointers, no aliasing, no exception handlers. Tasking (or restricting tasking as in Ravenscar profile) are outside the scope of this internship.

In the definition of the Profile we respected the 3 following rules:

1. First rule: we follow the order of Ada BNF to do the comparison. More precisely, we list all Ada language sections and subsections as in Ada LRM. If there is no difference between the current subsection and the corresponding subsection in SPARK LRM, we simply mention “No differences”. At the level of sub-subsections, only those which differ between SPARK and Ada are mentioned. To look more precisely at the exact differences between the two BNFs, we give in Annex A the full comparion between Ada BNF (syntax colored) and SPARK BNF.

2. Second rule: we define 2 scopes in our definition and implementation:

(a) The first scope is the scope of this internship. This corresponds to new extensions by translation realized in Sparkify during this internship or as part of the future Ada to SPARK translation tool (gnat2spark, the compiler gnat backend). For BNF differences in this scope, we explain the extension by translation relative to SPARK and/or the restriction relative to Ada if any.

(b) The second scope is the scope of possible extensions beyond the first scope. It contains in particular all extensions of SPARK already considered in the SPARK Pro roadmap.

3. We used a tri-colored Ada syntax to make the following distinction :

▷ Red: set of Ada syntax productions not to be included in the Verification Profile for Ada. These *restrictions* of the Profile in relation of Ada correspond of Ada entities not suitable for formal verification. They are summarized in section 2.3.4.

▷ Green: set of Ada syntax productions that are included in internship scope. This accounts for our proposed set of *extensions* in the Profile, a larger subset of Ada than SPARK . Namely, the set of Ada constructions we propose to translate into SPARK (summarized in section 2.3.3). The full design of some extensions added in Sparkify tool during our internship is given in chapter 3.

▷ Blue: set of Ada syntax productions that should be added to the profile in its final phase, that is, outside the scope of this internship. This set includes also SPARK syntax extensions already decided in the SPARK Pro roadmap [2].

The comparison between BNFs starts at section 2 since section 1 only contains general considerations.

▷ **2. Lexical Elements**

2.1 Character Set

Minor lexical differences. Ada uses the Universal Character Set (ISO/IEC 10646:2003) whereas SPARK uses only the full Latin-1 set. In the Profile we should use the richer Ada character set; consequently we should add an Extension_Of_Character_Set.

Suggested translation: beyond the scope of the internship, might be included in SPARK roadmap.

2.2 Lexical Elements, Separators, and Delimiters

No differences.

2.3 Identifiers

No major differences. In SPARK the identifiers predefined in its standard package version may not be re-declared whereas in Ada the identifiers predefined in Ada package standard may be re-declared (at user risk). We should keep this restriction of SPARK in the Profile; consequently, in relation to Ada we should define a restriction No_Predefined_Identifier_Definition.

In addition, the identifiers ASCII, Wide_Character, Wide_String, Constraint_Error, Program_Error, Storage_Error, and Tasking_Error which occur in the Ada version of package Standard plus the identifiers Universal_Integer, Universal_Float and Universal_Fixed may not be used in SPARK. In our Profile, may be some of these identifiers could be used.

2.4 Numeric Literals

There are two classes of numeric literals in Ada: real literals and integer literals. Literals may be

[2]Roadmap means that the overal extension will be done in the SPARK language itself.

expressed in decimal form, or based form (e.g. based 2 or 16). In SPARK, the based form is only permitted for integer literals, not for real literals. In the Profile we should lift this restriction by Extension_Of_Based_Form_In_Real_Literal.
Suggested translation: This extension is possible but beyond the scope of the internship, might be included in SPARK roadmap.

2.4.2:
based_literal ::=
    base # based_numeral [.based_numeral] # [exponent]

2.5 Character Literals

No syntactic differences. In Ada a character literal is an enumeration literal of a character type.
See BNF 3.5.2 for the semantic difference.

2.6 String Literals

No syntactic differences. In Ada a string literal is formed by a sequence of characters enclosed between two quotation marks used as string brackets. String literals are used also to represent operator symbols (see BNF 6.1) and array sub aggregates (see BNF 4.3.3). A semantic difference between Ada and SPARK is that concatenation operator & is applied dynamically in Ada whereas it is forbidden in SPARK. We include this dynamic feature of Ada in the Profile with the extension Extension_Of_Dynamic_Concatenation.
Suggested translation: for instance we can try to translate X & Y by a function *concat* (**in** X, **in** Y, **out** Z) that store the X elements in Z and then the Y elements in Z. For semantic validity we require as function contract, precisely a postcondition:
    ∀ I in X'Range, Z(I)=X(I) and ∀ J in Y'Range, Z(J + X'length)=Y(J).

2.7 Comments

A comment in Ada is written as an arbitrary piece of text following two minus signs. Thus – *this is a comment* and –# *is also another comment* are both valid Ada comments but only the last one corresponds to an annotation in SPARK. In our Profile, comments are like in Ada.

2.8 Pragmas

In Ada, a pragma is a compiler directive. Among them, there exists are language-defined pragmas (built-in) that give instructions for optimization, listing control, etc. An implementation may support additional implementation-defined pragmas (user defined). Pragmas are allowed at many places in Ada, while in SPARK their usage is restricted, so we propose the extension Extension_Of_Pragma.
Suggested translation: during preprocessing phase we should tanslate while putting in right places within SPARK the supported pragmas and rejecting unsupported pragmas. In our profile we propose to define a new pragma restrictions (see 2.3.4) to keep SPARK limitations not suitable for automatic formal verification.

2.9 Reserved Words

The Profile has the same reserved words as Ada. SPARK additional reserved words should be allowed as user identifiers.

▷ **3. Declarations and Types**

3.1 Declarations

A declaration is an Ada language construct that associates a name with (a view of) an entity definition. There are several forms of declaration in Ada defined as a basic_declaration.

```
3.1:
basic_declaration ::=
      type_declaration
    | subtype_declaration
    | object_declaration
    | number_declaration
    | subprogram_declaration
    | abstract_subprogram_declaration
    | null_procedure_declaration
    | package_declaration
    | renaming_declaration
    | exception_declaration
    | generic_declaration
    | generic_instantiation
```

In Ada, an abstract subprogram is a subprogram that has no body, but is intended to be overridden at some point when inherited. SPARK does not have abstract_subprogram_declaration. consequently we add an Extension_Of_Abstract_Subprogram_Declaration in the Profile. See the Suggested translation in its production of BNF 3.9.3.

In Ada, a null procedure declaration provides a shorthand to declare a procedure with an empty body. SPARK disallows this because in SPARK programming style it make no sense to have a null procedure. This can be allowed because for verification purpose we can translate it (see BNF 6.7).

The extension concerning generic constructions (Extension_Of_Generic) is beyond our internship's scope. In SPARK there are no generic declarations (not allowed in BNF 12.1). Generic instantiation (defined in BNF 12.3) is allowed simply for the predefined generic function unchecked conversion.
Suggested translation: This extension, already present in the SPARK Roadmap, is considered as important, as it would allow adding generic packages (generic containers) to the Standard Library. One idea of translation could be to proceed in a similar way as for regular packages, in order to verify their body. Formal parameters of the generic should be treated as normal declarations, with a dummy record created for formal private type declarations.

One considered restrictions in relation to Ada concerns exception declaration (not defined in SPARK BNF 11.1). Since exception-handling seriously complicates the sequential part of Ada we keep this limitation of SPARK; but declaring and raising exception is accepted (Extension_Of_Exception Declaration with the restriction No_Exception_Handlers define in section 2.3.4).
Suggested translation: translate exception declaration (Error : **exception**;) to a function. Thus we can call this function in the place of exception raising (**raise** Error). But as in SPARK, we disallow the response to one or more exceptions specified by an exception handler (**when** E : Error => Put("Bad exception handler : "); **raise**;).

Finally, subprogram_declaration, package_declaration and renaming_declaration are not considered

to be basic declarations in SPARK in order to facilitate visibility and refinement of entities. We should lift this restriction in the Profile.
In fact, in SPARK, subprogram declarations are actually still permitted in the visible and private parts of packages (see BNF 7.1). They are also permitted in declarative parts when immediately followed by the pragma Import (see BNF 3.11 and B.1 in LRM). Package declarations are still permitted in declarative parts (see BNF 3.11). Also renaming declarations are not considered to be basic declarations in SPARK. They are still permitted in the visible parts of packages (see BNF 7.1), and in declarative parts (see BNF 3.11), but are subject to restrictions in applications (see BNF 8.5).

3.2 Types and Subtypes

In Ada a type is characterized by a set of values, whereas a subtype of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype.

3.2.1 Type Declarations

Types are grouped into categories. The type definition is the actual type being defined, while the type declaration is a name and a type definition.

In Ada we can declare entities in an incomplete view known as incomplete_type_declaration, but they are actually only useful for pointers (access types in Ada parlance). We do not include them in the Profile because access type are excluded. Indeed an important difference between Ada and SPARK is that access types are excluded in SPARK. This is fine for our Profile because pointers really make verification intractable. Thereby No_Access_Types is a big restriction in the Profile.

3.2.1:
type_declaration ::=
    full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration

In Ada a composite type (other than an array or interface type) can have discriminants, which parametrize the type. SPARK forbids discriminated type declarations, partly because they are used for dynamic storage allocation (see definition in BNF 3.7). The only forms of discriminants supported by SPARK are task (see BNF 9.1 Ravenscar Profile) and protected type discriminants (see BNF 9.4). Consequently we define Extension_Of_Discriminant_Types.
Suggested translation: beyond our scope, included in SPARK roadmap.

3.2.1:
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] **is** type_definition;
  | task_type_declaration | protected_type_declaration

In Ada an interface type is an abstract tagged type, used with the abstract subprogram (see previously in 3.1). It provides a restricted form of multiple inheritance. This is not defined in SPARK because SPARK don't have abstract subprogram. Allowing interface type (Extension_Of Interface_Types) gives more expressivity, see its production in BNF 3.9.4.

3.2.1:
type_definition ::=
enumeration_type_definition
| integer_type_definition
| real_type_definition
| array_type_definition
| record_type_definition
| access_type_definition
| derived_type_definition
| interface_type_definition

SPARK disallows the use of derived types because it seriously complicates the formal definition of the language, and causes overloading whereas Ada allow them (defined in BNF 3.4) which is useful for user own explicit declaration. A lack of derived type on the scalar type limits user expression although it can be resolved using the subtype definition in Extension_Of_Derived_Scalar_Types. The suggested translation is explained in BNF 3.4 where the production of derived_type_definition is gived.

3.2.2 Subtype Declarations

A subtype indication is the view following reserved word "is" of subtype declaration (**subtype** Counter **is** Integer **range** 1 .. Integer'Last), its goal is to create a new subtype.

3.2.2:
subtype_indication ::=
[null_exclusion] subtype_mark [constraint]

No null exclusion in the Profile as a consequence of restriction No_Access_Types.

An important semantic difference between SPARK and Ada is that in SPARK, constraints shall be statically determinable and thereby variables and function calls cannot contain dynamic constraint. For instance the following expressions **function** Func_Int(Object) **return** Integer; **subtype** Counter **is** Integer **range** 1 .. Func_Int(A); are not allowed in SPARK . We want to lift this restriction with Extension_Of_Dynamic_Subtypes.
Suggested translation: this extension of SPARK is already included in SPARK roadmap, beyond our scope.

SPARK does not have digits constraints or delta constraints because they may reduce accuracy of the subtype or because SPARK does not have a decimal fixed point type as we will see in BNF 3.5.9.

3.2.2:
scalar_constraint ::=
range_constraint | digits_constraint | delta_constraint

3.3 Objects and Named Numbers.

In SPARK, in both constant and variable declarations, the nominal subtype shall be given by a subtype mark and shall not be unconstrained. That means SPARK does not allow anonymous types whereas Ada allows them in many places.

3.3.1:
object_declaration ::=
    defining_identifier_list : [aliased] [constant] subtype_indication [:= expression];
  | defining_identifier_list : [aliased] [constant] access_definition [:= expression];
  | defining_identifier_list : [aliased] [constant] array_type_definition [:= expression];
  | single_task_declaration | single_protected_declaration

In the profile we allow anonymous types (Extension_Of_Anonymous_Types) directly in object declaration (with subtype_indication, e.g: Var_Out : Integer **range** 0 .. 10 := 0; or array array_type_definition precisely in discrete_subtype_definition (see BNF 3.6), e.g: My_Array1 : **array** (1.. 10) of Integer;).
Suggested translation: consists in translating the uses of anonymous types in Ada into the uses of named types in SPARK, with a creation of the new named types.

In Ada initialized values can be general expressions whereas in SPARK the expression initializing an object shall not contain any of the following constructs :
· a name denoting an object which has not been declared by a constant or named number declaration;
· a function call which is not a call of a predefined operator, attribute or static function
· an indexed component;
· a selected component whose prefix denotes a record object.

In the Profile we would like to remove these restrictions especially in local variable declaration. In fact, SPARK disallows side-effects in functions and also in expressions within both local and global variables. The limitation on global variable usage can be justified but side-effects on local variable and expression should be allowed because this does not really entail verifiability problems.

Other possible extensions beyond our scope are Extension_Of_Single_Task_Declaration and Extension_Of_Single_Protected_Declaration. Indeed, in SPARK every entity has a name so single_task_declaration and single_protected_declaration which defines respectively an anonymous task type and anonymous protected type are disallowed.
We can authorize this extension simply by giving the name like we did Extension_Of_Anonymous_Types.
Suggested translation: beyond our scope because they concern task definition that we do not consider in this work.

3.4 Derived Types and Classes

SPARK does not have derived type definitions other than record type extensions (define in SPARK syntax rule 3.2.1). As we have seen in BNF's rule 3.2.1 this limitation in scalar type of SPARK can be lift as (Extension_Of_Derived_Scalar_Types).
Suggested translation: This limitation can be lifted simply by translating a derived scalar type in Ada (**type** Counter **is new** Integer **range** 1 .. 10;) into a subtype in SPARK (**subtype** Counter **is** Integer **range** 1 .. Integer'Last). This is semantically valid because statically, that means that the set of possible values of the derived type is a copy of the set of possible values of the parent type (i.e : *Counter is a copy of Integer* but the declaration *V:Counter;* is checked statically at compile time that *V is not of type Integer*. For a scalar type, the base range of the derived type is the same as that of the parent type so dynamically the checking of range constraint of a derived_type_definition is equivalent to checking subtype constraint.

3.4:

derived_type_definition ::=
  [abstract] [limited] new *parent*_subtype_indication [[**and** interface_list] record_extension_part]

3.5 Scalar Types:
In Ada scalar types comprise enumeration types, integer types, and real types. Enumeration types and integer types are called discrete types; each value of a discrete type has a position number which is an integer value. Integer types and real types are called numeric types. All scalar types are ordered, that is, all relational operators are predefined for their values. The difference between Ada and SPARK is that in SPARK the range constraint shall be static, crucial limitation to lift in the Profile with Extension_Of_Range_Constraint_In Discrete_Subtype.
Suggested translation: this extension is a prolongation of the previous extension of anonymous type; since translating anonymous subtypes with constraints into SPARK is an important ambition, and we are not concerned here with non-null exclusion (for access types), the only difference that remains is to allow ranges in discrete subtype definitions. This would allow to have static constraint in scalar type (e.g : Ten_Characters : String(1 .. 10)) or to insert automatically type in for-loop (e.g : **for** J **in** Buffer'Range **loop**).

3.5:
range_constraint ::=
  range range

3.5.1 Enumeration Types
In Ada, an enumeration type is defined by an enumeration type definition, where the enumeration literal specification is a parameterless function, whose defining name is the defining identifier or defining character literal in Ada. In SPARK, enumeration literals are not regarded as parameterless functions, but simply as names denoting the distinct values of the associated enumeration type. Other point, if the same enumeration literal is specified in more than one enumeration type definition, both directly visible at any point, the corresponding enumeration literals are said to be overloaded. This is forbidden in SPARK (Enumeration literals shall not be overloaded). In the Profile the Extension_Of_Enumeration_Literals_Overloading is possible.
Suggested translation: renaming the overloaded enumeration literal at any place where an overloaded enumeration literal occurs in the text of a program.

3.5.1:
enumeration_literal_specification ::=
    defining_identifier | defining_character_literal

3.5.2 Character Types
In Ada, an enumeration type is said to be a character type if at least one of its enumeration literals is a character literal. In SPARK is not possible to declare other character type and we don't have type Wide_Character. As we had seen already in enumeration type it is possible to lift this limitation of SPARK, by it is beyond our scope.

3.5.3 Boolean Types
Although the type Boolean is considered to be an enumeration type whose literals are simply True and False, it is possible to declare a full range subtype by writing :
subtype Valve_Open **is** Boolean;

but we cannot write for example :
subtype Valve_Open **is** Boolean **range** Boolean'First .. Boolean'Last**;**
nor
Subtype Always **is** Boolean **range** True .. True;
We keep this rules on Boolean Types : No_Boolean_Subtypes.

3.5.4 Integer Types

In Ada, "Modular_type_definition" is included in "integer_type_definition" unlike in SPARK where modular types is included directly in type definition on the BNF 3.2.1 because it follow some restrictions that we keep in the Profile.

· The Modulus of a type must be a positive power of 2. We keep the Binary_Modulus in the Profile.

· Unary arithmetic operators (unary -, +, abs) are not permitted. The unary "not" operator is allowed, as are all binary arithmetic and logical operators. This is a coding rule than a real check, thus we should allow Extension_Of_Unary_Modular_Operator.

3.5.4:
integer_type_definition ::=
    signed_integer_type_definition | modular_type_definition

SPARK does not have decimal fixed point types may be it reduce accuracy of subtype. For moment we keep this limitation of SPARK and we define a restriction No_Decimal_Fixed_Point.

3.5.9:
fixed_point_definition ::=
    ordinary_fixed_point_definition | decimal_fixed_point_definition

3.6 Array Types

An array object is a composite object consisting of components which all have the same subtype. As we have already announced in BNF 3.3 in SPARK a discrete subtype definition for the index and a component definition shall be a subtype mark only, and not specified in terms of an anonymous subtype. This is the important limitation to lift in discrete subtype definition.

3.6:
discrete_subtype_definition ::=
    *discrete*_subtype_indication | range

We lift this limitation also in a component definition.

3.6:
component_definition ::=
    [aliased] subtype_indication | [aliased] access_definition

3.6.1 Index Constraints and Discrete Ranges

3.6.1:
index_constraint ::=
    (discrete_range {, discrete_range})

3.6.1:
discrete_range ::=
    discrete_subtype_indication | range

3.7 Discriminants
As we have seen in BNF 3.2.1 some discriminant kind like discriminant type can be useful in Profile so we should allow known discriminant part in the Extension_Of_Discriminant_Types.

3.7:
discriminant_part ::=
    unknown_discriminant_part | known_discriminant_part

3.8 Record Types
SPARK and Ada record type definitions differ in different respects. It is interesting to give different aspect noticed in SPARK LRM :

Rule 1 : A record type definition shall not contain the reserved words abstract or limited. We lift this restriction because it is just a compilation constraint.

3.8:
record_type_definition ::=
    [[**abstract**] tagged] [**limited**] record_definition

Rule 2: In SPARK a component list cannot be the reserved word null unless the record is tagged. We should insert automatically the reserved word tagged to create a tagged type and thus components will be added by each extension; this should be syntaxically correct in verification purpose not in execution.

3.8:
record_definition ::=
      record
        component_list
      end record
    | **null record**

Rule 3: A record definition cannot be null record unless it is tagged . Same remark as a rule 2 because a record_definition of null record is equivalent to record null; end record.
Rule 4: A component list cannot have a variant part hence the "variant_part" (see BNF 3.8.1) is not defined. This limitation is a consequence of no discriminant in SPARK. Since discriminant type is one of extension of SPARK Pro roadmap. The extension of variant part in record is also possible.

3.8:
component_list ::=
      component_item {component_item}
    | {component_item} variant_part
    | **null**;

Rule 5 : A component item cannot be a representation clause. We keep this restriction in the Profile. In relation with Ada we should define No_Aspect_Clause_In_Record.

3.8:
component_item ::=
    component_declaration
  | aspect_clause

Rule 6 : A component declaration cannot have a default expression. This limitation should be useful to lift but it should be difficult to track the data flow of this kind of expressions because the values appear remote from where they are used. Consequently, in our Profile we keep this limitation of SPARK with the restriction No_Default_Expression_In_Record.

3.8:
component_declaration ::=
  defining_identifier_list : component_definition [:= default_expression];

As we have seen previously in array definition a "component_definition" in record definition must be a subtype mark, we lift this restriction by allowing anonymous type in record component definition.

3.8.1 Variant Parts and Discrete Choices

Variant parts, as we see previously should be supported in the Profile.
Discrete choices in both Ada and SPARK are used in aggregates and case statement alternatives although the category does not include the choice **others** in SPARK, which is instead directly incorporated into the syntax for array aggregates (see SPARK BNF 4.3.3) and case statements (see BNF 5.4).

3.8.1:
discrete_choice ::=
  expression | discrete_range | **others**

3.9 Tagged Types and Type Extensions

In Ada every type extension is a tagged type, and is a record extension or a private extension of some other tagged type, or a non-interface synchronized tagged type. In SPARK, the only extension type allowed is a tagged type of a record extension (define in BNF 3.2.1).

As we have already seen in 3.1 we should allow abstract subprogram by
Extension_Of_Abstract_Subprogram_Declaration.
Suggested translation: In verification context (not execution) we should handle the abstract subprograms with the regular subprogram. And also, this can be hidden by the hidden text directive of SPARK Examiner.
3.9.3:
abstract_subprogram_declaration ::=
  [overriding_indicator] subprogram_specification is abstract;

As we have already discussed in BNF 3.2.1 interface type can be translate.
Suggested translation: We should translate a interface type (e.g : **type** Queue **is limited interface**;) to a regular type used directly in the regular type subprogram translated to a abstract subprogram

(e.g: **procedure** Append(Q : **in out** Queue; Person : **in** Person_Name) **is abstract**;).
3.9.4:
interface_type_definition ::=
    [limited | task | protected | synchronized] interface [and interface_list]

3.10 Access Types
As we noticed in BNF's rule 3.2.1 No access Type because pointers really make verification intractable. In fact, a value of an access type provides indirect access to the object or subprogram it designates which can involve aliasing, which is extremely difficult for verification.

3.11 Declarative Parts
Ada define declarative part like

3.11:
declarative_part ::=
    {declarative_item}

In SPARK there are an additional production : "embedded_package_declaration" and "external_subprogram_declaration" because subprogram declarations are not allowed in a declarative part (except when immediately followed by pragma Import - see Annex B.1) but subprogram bodies are permitted (see in Annex declarative_item). This limitation imposes strict order of dependencies to prevent mutual recursion and facilitate refinement rules. Consequently we define the Extension_Of_Dependencies_Order.
Suggested translation: Modify the generation of SPARK code to create 3 SPARK packages for each Ada package P:
- a data package "P" which holds only data and types (no body)
- an external package "External_P" which only declares subprograms (no body)
- an internal package "Internal_P" which declares/defines subprograms (implemented body)

The external package is used everywhere a subprogram or type or object, etc. from P's initial specification is used. In particular, the implementation of the internal package calls the subprograms from the external package. This actually "breaks" the possible recursions, so that they look like non-recursive calls in SPARK. The internal package is used to verify the initial P's implementation. It should not be referenced in other packages.
With this additional division, both external and internal packages refer to data in data packages, so there is no possible loop in the dependence graph from own variables. There is one additional division which will be necessary in the general case, which will consist in dividing the data package into a "spec" and a "body" part which correspond to those variables/types defined in the original package spec and body, as in general variables/types defined in package bodies can generate circular dependencies.

In SPARK public and private child packages follow complex visibility rules with respect to their parent package; and subprograms bodies can only refer to subprograms already defined. This extension is named Extension_Of_Visibility_Of_Child_Packages.

Another difference in declarative_Item between both language is the introduction in SPARK of a generic_function_instantiation, introduced to allow instantiation of Unchecked_Conversion some time ago but intended to refer to generic_instantiation (BNF 12.3). Similarly to what was done to break recursion, it seems possible to generate a set of packages for verifying separately the body of a generic package and its various instances.

The main difference in visibility of usage of package is that SPARK forbids use_clause because obscure the origin of variables and consequently make programs hard to read and understand. We lift this restriction in the profile, Extension_Of_Use_Clause (see in BNF 8.4).

3.11: basic_declarative_item ::=
basic_declaration | aspect_clause | use_clause

Aspect clause is just a representation and operational items used to specify aspects of entities. No syntactic difference, SPARK just renames aspect_clause at "representation_clause" in BNF 13.1

No syntactic difference in proper_body. Just that production position of task body an protected body is difference in SPARK because task is added after in SPARK in Ravenscar profile.

3.11:
proper_body ::=
subprogram_body | package_body | task_body | protected_body.

We focus now to the different forms of names and expressions, and to their evaluation in the two BNFs in order to define the corresponding Profile name and expression.

▷ **4. Names and Expressions**

4.1 Names
According to Ada reference names can denote declared entities, whether declared explicitly or implicitly. Referencing does not apply in the Profile because there are No_Access_Types, so no explicit dereference and no implicit dereference.

4.1:
name ::=
direct_name
| explicit_dereference
| indexed_component
| slice
| selected_component
| attribute_reference
| type_conversion
| function_call
| character_literal
| qualified_expression

4.1:
prefix ::=
name | implicit_dereference

Since in SPARK new operators cannot be declared and operators are always called with infixed notation operator symbols are not direct name, there are primary. In the Profile overload operator should be allowed.

4.1:
direct_name ::=
    identifier | operator_symbol

A slice denotes a one-dimensional array formed by the sequence of consecutive components. It's sometime much use by user, disallowing slices as in SPARK limits user expression. So we should allow this Extension_Of_Slicing in the Profile.
Suggested translation: For instance from of Ada declaration *Stars : String(1 .. 120) := (1 .. 120 => '*');* we can use the for loop statement to define a a bounded array or a bounded slice of 15 characters *Stars(1 .. 15)*?

4.1.2: Slices
slice ::=
    prefix(discrete_range)

4.1.3 Selected Components
Selected_components are used to denote components; they are also used as expanded names called selector name.In SPARK, since we do not have overloading, a selector name cannot be a character_literal or an operator_symbol. But since we have the intention to allow overloading of enumeration literal, subprograms and operators this limitation will not make sens.

4.1.3:
selector_name ::=
    identifier | character_literal | operator_symbol

4.1.4 Attributes
An attribute is a characteristic of an entity that can be queried via an attribute_reference (prefix'attribute_designator) or a range_attribute_reference (prefix'range_attribute_designator). As in SPARK, the Profile syntax for attribute designators excludes the alternative Access since the corresponding attribute (link to access type) is not supported.

4.1.4:
attribute_designator ::=
    identifier[(static_expression)] | Access | Delta | Digits

4.2 Literals
A literal is either a numeric_literal, a character_literal, the literal null, or a string_literal.
For numeric_literal (see BNF 2.4) there are not difference.
For character_literal (see BNF 2.5 and 3.5.2)
For literal null : The literal null does not exist in SPARK, nor in the Profile because refere to pointer.
For string_literal : In Ada expected type for a primary that is a string literal (sequence of characters enclosed between two quotation marks) shall be a single string type. As SPARK has not Wide_Character consequently it has not also Wide_String.

4.3 Aggregates : An aggregate combines component values into a composite value of an array type, record type, or record extension. An aggregate is not a primary in SPARK (see Annex SPARK rule 4.4), which implies that whenever an aggregate is used in an expression it must be qualified by an appropriate type mark to form a qualified (Extension_Of_Type_Qualifier_On_Aggregate).
Suggested translation: This extension consists in computing Ada's aggregate without type qualifiers into SPARK's aggregate profile, where only aggregates that are prefixed with a type qualifier are allowed. It is to generate automatically the needed type qualifiers before aggregates to avoid users insert these manually.

```
4.3:
aggregate ::=
    record_aggregate | extension_aggregate | array_aggregate
```

4.3.1: Record Aggregates
In a record aggregate, SPARK and Ada syntax differ in many respects. In Ada we have the syntax below :

In the SPARK syntax, positional and named component associations shall not be mixed within the same record aggregate. This limitation can be lift but we can keep this restriction in our Profile for good programming style.

```
4.3.2:
record_aggregate ::=
    (record_component_association_list)
```

SPARK does not permit the reserved words null record in an aggregate unless it is an extension aggregate. We should lift this limitation until we handle null record but as it is already included in extension aggregate (BNF 4.3.2) this right.

```
4.3.1:
record_component_association_list ::=
     record_component_association {, record_component_association}
    | null record
```

```
4.3.1:
record_component_association ::=
     [component_choice_list =>] expression
    | component_choice_list => <>
```

In Ada a record aggregate, each named component association can have many aggregate choices, and others can be used, that is impossible in SPARK we must have only aggregate choice and others cannot be used. This restriction should be lifted in the Profile simply by changing many aggregate choices by only one.

```
4.3.1:
component_choice_list ::=
     component_selector_name {| component_selector_name}
```

| others

4.3.2: Extension aggregates

In Ada, an extension_aggregate specifies a value for a type that is a record extension by specifying a value or subtype for an ancestor of the type.Extension aggregates in SPARK can have null record. This compensates record aggregates without null record, see above in BNF 4.3.1.

4.3.2:
extension_aggregate::=
  (ancestor_part **with** record_component_association _list)

In SPARK the ancestor part of an extension aggregate may not be a subtype mark. We should lift this limitation.

4.3.2:
ancestor_part ::=
   expression
  | subtype_mark

4.3.1: Array Aggregates

Contrary to record component Ada already forbids this mixing positional and named component associations for array aggregates. This is fine for good programming style.

4.3.3:
array_aggregate ::=
  positional_array_aggregate | named_array_aggregate

In SPARK, the number of components in a positional array aggregate, or the range of choices in a named array aggregate, must be (statically) consistent with the bounds associated with the qualifying type mark. wee keep this restriction in the Profile.

4.3.3:
positional_array_aggregate ::=
   (expression, expression {, expression})
  | (expression {, expression}, **others** => expression)
  | (expression {, expression}, **others** => <>)

4.4 Expressions

An expression is a formula that defines the computation or retrieval of a value.
Ada and SPARK expressions differ in minor the following respects:

1. Null is not a primary
2. Allocators are not primaries because Allocators are not allowed in SPARK because referred to access type : so no allocator in the Profile.
3. Aggregates are not primaries because SPARK does not allow aggregates in expressions; it is why since we had seen in BNF 4.3 whenever an aggregate is used in an expression it must be qualified by an appropriate type mark to form a qualified.

4. Character literals and type conversions are primaries. This rule is required in SPARK because character literals and type conversions are not names in SPARK. This rule make no sens in our Profile because we want to allow overloading.

4.4:
primary ::=
    numeric_literal
  | null
  | string_literal
  | aggregate
  | name
  | qualified_expression
  | allocator
  | (expression)

4.5 Operators and Expression Evaluation

Ada language defines additional the logical_operator and highest_precedence_operator in order of increasing precedence. In SPARK, the logical operators and, or and xor for one-dimensional arrays of Boolean components are defined only when both operands have the same upper and lower bounds.
4.5:
logical_operator ::=
  and | or | xor

4.5:
highest_precedence_operator ::=
  ** | abs | not

4.6 Type Conversions

The operand of a type_conversion is the expression (value conversions) or name (view conversions) within the parentheses. Ada has value conversions and view conversions on arrays and objects of numeric types whereas in SPARK such conversions are always value conversions. In reality, view conversions only occur with tagged types. Thus, Extension_Of_View_Conversion should be allowed as we should allow some tagged types.
4.5:
type_conversion ::=
  subtype_mark(expression) | subtype_mark(name)

4.7 Qualified Expressions

A qualified expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate. There are no syntactic differences between a qualified expression in Ada and in SPARK. Nevertheless there are semantic difference already discussed in 4.3 and 4.4.

4.8 Allocators

Already seen in 4.4 no allocator as consequence on No_Access_Types.

4.9 Static Expressions and Static Subtypes

Static means determinable at compile time, using the declared properties or values of the program entities. In Ada, some expressions of a scalar or string type are defined to be static (see the LRM). In SPARK the definition of static expression is extended to include enumeration literals explicitly. A SPARK program shall not contain a static expression whose value violates a range constraint or an index constraint. This limitation of SPARK should be extended as we have already seen previously points.

▷ 5 Statements

5.1 Simple and Compound Statements - Sequences of Statements

A statement is either simple or compound. A simple statement encloses no other statement. A compound statement can enclose simple statements and other compound_statements.

In SPARK we have a new terminal return_statement (in BNF 6.5) which come with the new restriction return statements appear once and only once as a last statement in function subprogram. The restriction No_Multiple_Return_Statement will be kept in the Profile. Although we could theoretically lift this limitation, but no interest for_the customers.

As in SPARK, in the Profile the are No_Goto_Statement (see BNF 5.8), No_Abort_Statement and No_Requeue_Statement, but we should keep raise statement because we hope to allow a simple exception scheme (see BNF 3.1).

5.1:
```
simple_statement ::=
      null_statement
    | assignment_statement
    | exit_statement
    | goto_statement
    | procedure_call_statement
    | simple_return_statement
    | entry_call_statement
    | requeue_statement
    | delay_statement
    | abort_statement
    | raise_statement
    | code_statement
```

In SPARK "accept_statement" and "select_statement" cannot be employed because they refer to entry, beyond internship scope. Block statement is useful to define unconstrained types in news scope during the programming Extension_Of_Block_Statement (see suggested translation in BNF 5.6)

Once block statement is allowed we should allow extended_return_statement (Extension_Of_Extended_Return_Statement).
Suggested translation: we should transform it to a simple return statement in a block statement.

5.1:
```
compound_statement ::=
      if_statement
    | case_statement
```

| loop_statement
| block_statement
| extended_return_statement
| accept_statement
| select_statement

5.2 Assignment Statement

There are some syntactic differences between Ada assignment_statement and SPARK. SPARK was added “unconstrained_array_assignment ” to allow in this way the one dimensional unconstrained assignment. This grammar excludes the possibility of multi-dimensional array aggregates. We keep SPARK restriction.

5.2:
assignment_statement ::=
  variable_name := expression

5.3 If Statements

No difference. If statements are exactly as in Ada.

5.4 Case Statements

No major difference. Just that in the SPARK grammar the syntactic category discrete_choice does not include the choice others, which is instead directly incorporated in the case statement syntax. The Profile used Ada grammar syntax.

5.4:
case_statement ::=
  case expression is
   case_statement_alternative
   {case_statement_alternative}
  end case;

5.5 Loop Statements

In Ada, a loop statement may be named by a loop statement identifier and the range in a for statement need not give the type explicitly (e.g : **for** I **in** 1..10 **loop**), but in SPARK a loop parameter specification shall include an explicit subtype mark for the range over which the loop parameter will iterate (e.g : **for** I **in** integer **range** 1..10 **loop**). This prevents the loop parameter from having an anonymous subtype. This extension (Extension_Of_Range_Constraint) consist simply to insert automatically the corresponding type.

5.5: loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] *discrete*_subtype_definition

5.6 Block Statements

A block_statement encloses a handled_sequence_of_statements optionally preceded by a declarative_part. In SPARK all subtypes are static and so inner blocks are really needed for dynamic declarations.
Suggested translation: this translation should consist in transforming any block statement of Ada by an

inlined procedure in SPARK for instance.

5.6:

```
block_statement ::=
    [block_statement_identifier:]
      [declare
        declarative_part]
      begin
        handled_sequence_of_statements
      end [block_identifier];
```

5.7 Exit Statements

Exit statements always apply to their very neighbor enclosing loops, so in SPARK loop_name is replaced by simple_name. In Ada, the named form of exit statement can transfer control out of an inner loop of a nested set of loops to the end of the outer loop with the given name. In the Profile we should define No_Outer_Loop_In_Exit_Statement

5.7:

```
exit_statement ::=
    exit [loop_name] [when  condition];
```

5.8 Goto Statements :

In Ada, a goto statement specifies an explicit transfer of control from this statement to a target statement with a given label. This makes verification of program very difficult to control. As SPARK, No_Goto_Statements in the Profile.

5.8:

```
goto_statement ::=
     goto label_name;
```

6 Subprograms

A subprogram is a program unit or intrinsic operation whose execution is invoked by a subprogram call. There are two forms of subprogram: procedures and functions. The definition of a subprogram can be given in two parts: a subprogram declaration defining its interface, and a subprogram_body defining its execution. Operators and enumeration literals are functions.

6.1 Subprogram Declarations

In Ada a subprogram declaration declares a procedure or function by a subprogram specification. There are no major differences in SPARK, except that procedure and function specification are treated distinctly in order to aid the discussion on annotations (for the additional rules in SPARK, see Annex in BNF 6.1.1). In fact, in Ada, the transactions between a subprogram are specified by the parameter modes, in SPARK, the transactions between subprograms and its environment are specified and controlled much more precisely, by adding annotations to subprogram specifications and imposing some additional rules.

Thus global and state annotation can be generated : Extension_Of_Global_And_State_Annotations.
Suggested translation: this translation consist in infering global and state annotations to detect the read and write effects. Precisely it collects declarations of global variables in package declaration and body,

and generate appropriate "own" and "initializes" annotations based on those. The precised data-flow should be implemented in the GNAT back end implementation that will follow the prototyping phase

6.1:
subprogram_declaration ::=
    [overriding_indicator] subprogram_specification;

As we had seen in BNF 3.11 declarative_part's rule, SPARK restricts where subprograms can be declared, we remove this restriction : Extension_Visibility_Of_Child_Packages.

Another big limitation of SPARK is that there are not subprograms overloading. We lift this limitation: Extension_Of_Subprograms_Overloading
Suggested translation: this extension is just a matter of renaming the subprograms to discriminate between overloaded ones. To detect which subprograms are overloaded, the easiest way used is to maintain a set of names of subprograms in scope, which is created afresh for each new unit, and augmented with all the subprograms of a declarative block each time a declarative block is entered

There are significant differences regarding subprogram specification in SPARK. One difference is that in SPARK a function designator must be an identifier (not an operator symbol). If we allow overloading of operator (see BNF 6.6), this restriction can be lifted.

6.1:
designator ::=
    [parent_unit_name . identifier] | operator_symbol

6.1:
defining_designator ::=
    defining_program_unit_name | defining_operator_symbol

Another difference is that SPARK prevents aliasing between parameters and global of subprograms because SPARK does not have access types. we keep this restriction : no aliasing between parameters and global.
A further difference is that functions can return unconstrained type in Ada, no in SPARK. We keep this limitation of SPARK: No_Unconstrained_Object_Returns.

6.1:
parameter_and_result_profile ::=
    [formal_part] return[null_exclusion] subtype_mark
    | [formal_part] return access_definition

In SPARK the default expressions are not allowed for formal parameters as they are in Ada. We lift this limitation by Extension_Of_Default_Expression_In_Parameter_Specification.
Suggested translation: this translation should consist to retrieve the default expression in the parameter specification and used it in the variable which used it.

6.1:
parameter_specification ::=

```
    defining_identifier_list : mode
  | [null_exclusion] subtype_mark  [:= default_expression]
  | defining_identifier_list :  access_definition  [:= default_expression]
```

6.2 Formal Parameter Modes

No syntactic differences. But, the rules of SPARK, in particular the rules to prohibit aliasing in the execution of procedures (see BNF 6.4), prevent the possibility of assigning to an object via one access path and then reading its value via a distinct access path. This ensures that the effect of the program will not depend on whether the parameter is passed by copy or by reference.

6.3 Subprogram Bodies

As a declaration of a procedure subprogram, their bodies may contain a procedure annotation

```
6.3:
subprogram_body ::=
  [overriding_indicator]
    subprogram_specification is
      declarative_part
    begin
      handled_sequence_of_statements
    end [designator];
```

A designator shall appear at the end of every subprogram body (repeating the designator of the subprogram specification). We should lift this limitation.

6.4 Subprogram Calls

A procedure call is a statement; a function call is an expression and returns a value.
A subprogram declared in a package shall not be called in a private child of that package or in any descendant of such a private child. These rules, together with the rules seen in BNF 3.11 imply that subprograms cannot be called recursively.This limitation will be lifted.

```
6.4:
procedure_call_statement ::=
    procedure_name;
  | procedure_prefix  actual_parameter_part;

function_call ::=
    function_name
  | function_prefix  actual_parameter_part
```

In SPARK, positional and named parameter associations shall not both be used in the same subprogram call. We should lift this restriction, but for good programming style it is preferable to keep this restriction: No_Mixed_Positional_And_Named_Associations.

```
6.4:
actual_parameter_part ::=
   (parameter_association  {, parameter_association})
```

6.4:
parameter_association ::=
    [formal_parameter_selector_name =>] explicit_actual_parameter

6.5 Return Statements

The last statement in the sequence of statements of a function subprogram shall be a return statement in SPARK, we keep this restriction in the Profile (No_Multiple_Return_Statement, already seen in BNF 5.1).

6.5:
simple_return_statement ::=
    return [expression];

Extend return statement should be translate in simple return statement (Extension_Of_Extended_Return_Statement).

6.5:
extended_return_statement ::=
    return defining_identifier : **[aliased]** return_subtype_indication [:= expression] [**do**
        handled_sequence_of_statements
    end return];

6.6 Overloading of Operators

In Ada like in SPARK, renaming declarations are allowed for operators (see BNF 8.5) but in SPARK user-defined operators are not permitted.
We could allow it in the Profile : Extension_Of_Operator_Overloading.
Suggested translation: this extension should consist in renaming operators to discriminate between overloaded ones. This renaming just work simply as the solution proposed for Extension_Of_Subprograms _Overloading. This translation may be refine in the potential final solution.

6.7 Null Procedures

Extension_Of_Null_Procedure_Declaration : May be translated by the procedure declaration which has a null statement in his body.

6.7:
null_procedure_declaration ::=
    [overriding_indicator] procedure_specification **is null**;

▷ 7 Packages

Packages are program units that allow the specification of groups of logically related entities. SPARK has several important concepts associated with packages which do not exist in Ada. Here we have much additional production in SPARK because it has several important concepts associated with packages which do not exist in Ada. We present simply the productions that we will keep in the Profile, for more information to an additional concept refer to SPARK LRM.

7.1 Package Specifications and Declarations

Here we give only Ada package declaration, for more details on the difference for SPARK declaration refer to the Annex A.

```
7.1:
package_declaration ::=
    package_specification;
```

```
7.1:
package_specification ::=
    package defining_program_unit_name is
    {basic_declarative_item}
    [private {basic_declarative_item}]
    end [[parent_unit_name.]identifier]
```

In SPARK a package specification must end with the name of the package. we lift this limitation in the preprocessing phase if Ada code don't mention end name explicitly.

SPARK have an additional structure different to Ada in the many respects :

A package declaration contains an optional inherit clause (described in BNF section 7.1.1 below) before the package specification. We keep this characteristic in the Profile because this rule of inheritance provides a relatively simple and precise way of specifying, and controlling, the access to external entities. This enter in our profile with Extension_Visibility_Of_Child_Packages.

7.2 Package Bodies

As we have seen previously, the limitation that subprograms and package bodies must end with the name of the package is lifted. Another difference is that a package body may begin with a refinement definition (described in SPARK grammar in BNF 7.2.1).

```
7.2:
package_body ::=
    package body defining_program_unit_name is
       {declarative_part}
     [begin
        handled_sequence_of_statements]
     end [[parent_unit_name.]identifier]
```

7.3 Private Types and Private Extensions

In Ada, the declaration (in the visible part of a package) of a type as a private type or private extension serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself).
SPARK and Ada private type declarations differ in that a private type declaration in SPARK cannot have a discriminant part. beyond internship scope

```
7.3:
private_type_declaration ::=
```

type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**;

7.3:
private_extension_declaration ::=
type defining_identifier [discriminant_part] **is**
[**abstract**] [**limited** | **synchronized**] **new** ancestor_subtype_indication
[**and** interface_list] **with private**;

7.4 Deferred Constants
In Ada, deferred constant declarations may be used to declare constants in the visible part of a package, but with the value of the constant given in the private part. They may also be used to declare constants imported from other languages. SPARK does not permit a deferred constant declaration to be completed by a pragma Import. Beyond our scope.

7.5 Limited Types
A limited type is (a view of) a type for which copying (such as for an assignment_statement) is not allowed. A non limited type is a (view of a) type for which copying is allowed. In SPARK it is not defined. It is a compile time constraint, to be lifted

7.6 User-Defined Assignment and Finalization
In Ada, three kinds of actions are fundamental to the manipulation of controlled objects: initialization, finalization, and assignment. Every object is initialized, either explicitly or by default, after being created. Every object is finalized before being destroyed . An assignment operation is used as part of assignment_statements, explicit initialization, parameter passing, and other operations.
SPARK does not have controlled types and hence there are no user-defined initialization, assignment or finalization operations. This possible extension is beyond our scope.

▷ **8 Visibility Rules**

The rules defining the scope of declarations and the rules defining which identifiers, character_literals, and operator_symbols are visible at (or from) various places in the text of the program are described in this section.

8.1 Declarative Region
No difference.

8.2 Scope of Declarations
No difference.

8.3 Visibility
The rules defining the scope of declarations and the rules defining which identifiers, character_literals, and operator_symbols are visible at (or from) various places in the text of the program are described the visibility in Ada. The associations between declarations and occurrences of identifiers and the places where particular identifiers can occur are governed by the scope and visibility rules of Ada. SPARK has additional restrictions in the visibility that will be lifted in the Profile.

8.4 Use Clauses
In Ada, a use_package_clause achieves direct visibility of declarations that appear in the visible part of

a package; a use_type_clause achieves direct visibility of the primitive operators of a type. In SPARK use (package) clauses are not allowed. Rather than simplifying the use clause and reducing the contexts in which it can be placed, SPARK does not permit the basic use clause at all. SPARK allows the use type clause in certain contexts; use type clauses are subject to certain restrictions (see LRM). We extend this restriction in the Profile Extension_Of_Use_Clause.
Suggested translation: Use the prefixing notation, thus the entity X will be prefixed by it scope package P as P.X . The problem is that the prefix notation is prohibited in SPARK with operators (e.g : P."+" is illegal). The solution is the use type clause.

8.4:
use_clause ::=
    use_package_clause | use_type_clause

8.4:
use_package_clause ::=
    use package_name {, package_name};

8.5 Renaming Declarations
In Ada, a renaming_declaration declares another name for an entity, such as an object, exception, package, subprogram, entry, or generic unit. Alternatively, a subprogram_renaming_declaration can be the completion of a previous subprogram_declaration. In SPARK, the only renaming declarations are those for subprograms and (child) packages. As exception and generic is a potential future extension of SPARK the renaming these kind of entities is possible. We allow renaming object declaration with Extension_Of_Renaming.

8.5:
renaming_declaration ::=
      object_renaming_declaration
    | exception_renaming_declaration
    | package_renaming_declaration
    | subprogram_renaming_declaration
    | generic_renaming_declaration

8.6 The Context of Overload Resolution
As we have already seen subprograms, enumeration literals, character literals and string literals have a unique meaning in SPARK, by the rules of the language they cannot be overloaded is lifted.

▷ 9 Tasks and Synchronization

This section describes the various forms of tasks interactions . The execution of an Ada program consists of the execution of one or more tasks. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks. In SPARK, support for tasks and synchronization is based on the Ravenscar Profile (Ravenscar2003) beyond our scope. we interest sequential SPARK. No_Tasking

▷ 10 Program Structure and Compilation Issues

10.1 Separate Compilation

In Ada, a program unit is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units. In SPARK, a subprogram declaration or body is not a library item. The main program in SPARK is distinct from a subprogram body and is a library unit. It is distinguished by the presence of a main_program annotation. Thus the presence of an optional inherit clause and private package declarations are expressed differently in both Ada and SPARKsyntax (see Annex).

Since a consequence of as we have seen previously some program units do not allowed in SPARK may be allowed for verification purpose.

10.1.1:
library_item ::=
  [**private**] library_unit_declaration
  | library_unit_body
  | [**private**] library_unit_renaming_declaration

10.1.1:
library_unit_declaration ::=
  subprogram_declaration
  | package_declaration
  | generic_declaration
  | generic_instantiation

10.1.1:
library_unit_body ::=
  subprogram_body | package_body

In SPARK, a context clause contains with clauses and use type clauses "use_type_clause" only, and no use (package) clauses because all names should be prefixed with the packages they come from. This restriction is included on an Extension_Of_Use_Clause.

10.1.2:
context_item ::=
  with_clause | use_clause

10.1.2:
with_clause ::=
  limited_with_clause | nonlimited_with_clause

10.1.3 Subunits of Compilation Units

Subunits are like child units, with these (important) differences: subunits support the separate compilation of bodies only (not declarations); the parent contains a body_stub to indicate the existence and place of each of its subunits; declarations appearing in the parent's body can be visible within the subunits. The differences for subprogram body stubs for SPARK 2005 and Ada 2005 are identical to subprogram declarations – see Section 6.1.

10.1.3:
body_stub ::=
    subprogram_body_stub
    | package_body_stub
    | task_body_stub
    | protected_body_stub

10.2 Program Execution
No syntax define. An Ada program consists of a set of partitions, which can execute in parallel with one another, possibly in a separate address space, and possibly on a separate computer. In the profile we interest to determinism execution.

▷ 11 Exceptions

An exception represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an exception occurrence. To raise an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called handling the exception. As we have already seen we should allow exception declaration and raise exception name; but no exception handler (we can use GNAT pragma restriction No_Exception_Handlers).

11.1:
exception_declaration ::= defining_identifier_list : exception;

11.2:
exception_handler ::=
    when [choice_parameter_specification:] exception_choice {| exception_choice} => sequence_of_statements

11.3:
raise_statement ::=
    raise; | raise exception_name [with string_expression];

▷ 12 Generic Units

A generic unit is a template, which can be parametrized, and from which corresponding (non generic) subprograms or packages can be obtained.
As we have seen in 3.1 this extension is beyond our scope; already included in SPARK roadmap.

▷ 13 Representation Issues

The representation issues section describes features for querying and controlling certain aspects of entities and for interfacing to hardware.

13.1 Operational and Representation Items
No syntactic difference; just renaming Ada suntax word aspect_clause to representation_clause in SPARK. Representation clauses may appear in SPARK texts. The SPARK Examiner checks their

syntax, which must conform to the syntax rules given in Chapter 13 of the Ada LRM, but it ignores their semantics. A warning message to this effect is given whenever the SPARK Examiner encounters a representation clause.

13.1:
aspect_clause ::=
    attribute_definition_clause
    | enumeration_representation_clause
    | record_representation_clause
    | at_clause

13.3 Operational and Representation Attributes

The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate operational or representation attributes. Some of these attributes are specifiable via an attribute_definition_clause.

13.3:
attribute_definition_clause ::=
    for local_name'attribute_designator **use** expression;
    | **for** local_name'attribute_designator **use** name;

13.4 Enumeration Representation Clauses

No difference.

13.5 Record Layout

A record_representation_clause specifies the storage representation of records and record extensions, that is, the order, position, and size of components (including discriminants, if any).

In Ada the position is a static expression whereas in SPARK it is a static simple expression, this restriction should be lift.

13.5.1:
position ::=
    *static*_expression

13.8 Machine Code Insertions

13.8:
No syntactic difference. Just that code statements are permitted in SPARK subprogram bodies in Section 6.3 with a code insertion.

13.9 Unchecked Type Conversions

An unchecked type conversion can be achieved by a call to an instance of the generic function Unchecked-Conversion. SPARK recognizes the predefined generic function Unchecked_Conversion and allows instances of this. SPARK checks the static semantics of the instantiation but does not perform any of the dynamic semantic checks relating to the size and alignment of the actual subtypes used in the instantiation.

13.10 Unchecked Access Value Creation

The attribute Unchecked_Access is used to create access values in an unsafe manner — the programmer is responsible for preventing "dangling references." In the Profile there are not access type.

13.11 Storage Management

The package Ada.Finalization is not predefined in SPARK and there are no user-defined storage pools.

13.12 Pragma Restrictions

A pragma restrictions expresses the user's intent to abide by certain restrictions, this is useful to define our subset of Ada restrict compared to Ada and extend compared to SPARK. In the profile we can use the full set of restrictions allowed in a pragma restrictions of GNAT (All RM defined Restriction identifiers implemented in GNAT and GNAT additional restrictions provided). We provide also some restrictions. The additional restrictions we consider in the profile is listed in BNF 2.3.4.

13.12

```
restriction ::=
    restriction_identifier
    | restriction_parameter_identifier  =>  restriction_parameter_argument
```

13.13 Streams

A stream is a sequence of elements comprising values from possibly different types and allowing sequential access to these values. The package Ada.Streams is not predefined in SPARK and there are no stream types. This limitation should be lifted by SPARK container.

Finally in Ada summary BNF it redefine J.3 and J7 and J8 of Annex J Obsolescent Features of standard, whereas SPARK define new subsection for this characteristic (for instance J.7 is define in 13.5 in SPARK RM.

J.3

```
delta_constraint ::=
    delta static_expression [range_constraint]
```

J.7:

at_clause ::= for direct_name use at expression;
Already defined in BNF 13.5 in SPARK

J.8:

mod_clause ::= at mod static_expression;
Already defined in BNF 13.5.1 in SPARK.

In the following section, we sump up the Profile (the suggested canvas for Verification Profile for Ada) : the extensions proposed in the larger subset than SPARK and the restrictions of the Profile.

2.3.3 Extensions proposed in the Profile

This following set of extensions consists in lifting by translation some limitations of SPARK in order to provide a larger Verification Profile of Ada. Some of them are already included in SPARK roadmap (considered to be the real extensions of SPARK language), others is already realized in Sparkify (Ada to SPARK translator prototype tool) and all of them should be implemented in a potential final solution tool (backend of gnat2spark).

2.3.3.1 Extensions included in SPARK roadmap

Extension_Of_Generic:

> This extension is the big extension expected in SPARK language itself. It will change SPARK programming langage style and will provide additional packages in the Standard library (generic containers).

Extension_Of_Dynamic_Subtypes:

> This extension will consist in allowing dynamic subtypes (dynamic constraint) in SPARK.

Extension_Of_Discriminant_Types:

> This extension will consist in allowing discriminant types (see BNF 3.2.1), consequently, extension of variant part in record (see BNF 3.8) will also be allowed.

Extension_Of_Character_Set:

> This extension should consist in extending the character set of SPARK.

Extension_Of_Based_Form_In_Real_Literal:

> This extension should consist in allowing the based form also in real literals as it is already in integer literal in SPARK.

2.3.3.2 Extensions proposed by translation in the Profile

Extension_Of_Dynamic_Concatenation:

> This extension consist in allowing the dynamic concatenation operator of Ada in the Profile in addition of static concatenation already allowed in SPARK.

Extension_Of_Pragma:

> This extension consists in rejecting unsupported pragma for the moment, and should consist in placing correctly pragmas defined in many place in Ada in the right place in SPARK. We propose to define a new pragma restrictions (see 2.3.4) to set up SPARK restrictions making verification intractable.

Extension_Of_Abstract_Subprogram_Declaration:

This extension should consist in translating an abstract subprogram to a regular subprogram (procedure).

Extension_Of_Null_Procedure_Declaration:

This extension should consist in transforming a null procedure declaration into a procedure declaration that has a body.

Extension_Of_Exception_Declaration:

This extension should consist in allowing a declaration and raise exception with the restriction No_Exception_Handlers defined in restriction section 2.3.4.

Extension_Of_Interface_Types:

This extension should consists in translating a interface type to regular type.

Extension_Of_Derived_Scalar_Types:

This extension consists simply in translating a derived type in Ada into a subtype in SPARK.

Extension_Of_Anonymous_Types:

This extension consists in translating the uses of anonymous types in Ada into the uses of named types in SPARK, with a creation of the new named types. This extension is done in many places : directly in a definition of an object or a variable (for scalar or array type and for constant); in both the type of index and type of elements in an array type; in the type of a component in a record definition.

Extension_Of_Range_Constraint_In Discrete_Subtype:

This extension is a prolongation of the previous extension of anonymous type by allowing range in discrete subtype definition.

Extension_Of_Single_Task_Declaration:

This restriction should allow an anonymous task type by simply renaming the name of task type as it is doing in Extension_Of_Anonymous_Types.

Extension_Of_Single_Protected_Declaration:

This restriction should allow an anonymous single and protected type by simply renaming the name type.

Extension_Of_Enumeration_Literals_Overloading:

This extension consist in allowing overloading of enumeration literals by renaming this enumeration literal at any place where it is overloaded occurrence in the text of a program. This extension should also consist in allowing user-defining character literals in enumeration type definition.

Extension_Of_Subprograms_Overloading:

This extension is just a matter of renaming the subprograms to discriminate between overloaded ones. To detect which subprograms are overloaded, the easiest way used is to maintain a set of names of subprograms in scope, which is created afresh for each new unit, and augmented with all the subprograms of a declarative block each time a declarative block is entered.

Extension_Of_Operator_Overloading:

Handling overloading of operators should also be extended.

Extension_Of_Unary_Modular_Operator:

This extension consists in allowing unary arithmetic operators (unary -, +, abs) in SPARK.

Extension_Of_Type_Qualifier_On_Aggregate:

This extension consists in computing Ada's aggregate without type qualifiers into SPARK's aggregate profile, where only aggregates that are prefixed with a type qualifier are allowed. It is to generate automatically the needed type qualifiers before aggregates to avoid users insert these manually.

Extension_Of_Default_Expression_in_Parameter_Specification:

This extension should consist in allowing default expression in parameters of subprograms.

Extension_Of_Slicing:

This extension consist in translating array slicing in Ada into a bounded array in SPARK by a transformation with a loop statement.

Extension_Of_View_Conversion:

This extension should consist in allowing type conversions on tagged types where we interest to obtain a different view of object designated, not to convert explicitly an expression to a different type.

Extension_Of_Block_Statement:

This extension should consist in transforming any block statement of Ada by an inlined procedure in SPARK for instance.

Extension_Of_Extended_Return_Statement:

This extension should consist translating it to a simple return statement in a block statement.

Extension_Of_Global_And_State_Annotations:

This extension consist in infering global and state annotations to detect the read and write effects. Precisely it collects declarations of global variables in package declaration and body, and generate appropriate "own" and "initializes" annotations based on those.

Extension_Of_Use_Clause:

This extension consist in using the Ada prefixing notation to prefix all entities by its package name; and use the “use type clause” in certain contexts.

Extension_Visibility_Of_Child_Packages:

This extension consists in inheriting the parent package in a child package and prefix the global read/written from the parent in the child package global annotations.

Extension_Of_Dependencies_Order:

This extension consist in breaking order of dependency between packages and subprograms to allow recursion in SPARK.

Extension_Of_Renaming:

This restriction should consist in allowing renaming in SPARK useful for good programming skill.

2.3.4 Restrictions proposed in the Profile

These restrictions particularly correspond to the limitations of SPARK that make verification intractable. We suggested to keep these restrictions in our Profile as a potential additional pragma restrictions (see BNF 13.12); As in Ada RM H.4, these additional restrictions should facilitate the demonstration of program correctness or program verification.

No_Predefined_Identifier_Definition:

> This restriction ensures at compile time that user had not used the identifiers predefined in the package standard.

No_Exception_Handlers:

> Use GNAT restriction exception handlers.

No_Access_Types:

> This restriction ensures at compile time that no access types are used. Consequently we prohibit all others productions that referred to access type:
>
> Incomplete_Type_Declaration (see Ada RM 3.10.1)
>
> Explicit_Dereference (see Ada RM 4.1-8)
>
> Implicit_Dereference (see Ada RM 4.1-8) ,
>
> No_Allocators, already defined in Ada RM H.4 High Integrity Restrictions (restrictions clause to facilitate the demonstration of program correctness by allowing tailored versions of the run-time system)
>
> No_Aliasing_Between_Parameters_And_Global.

No_Tasking:

> This restriction ensures that there are no Ravenscar profile. We interest to verification of sequential SPARK; for Ravenscar this is possible job beyond our scope.

No_Unconstrained_Object_Returns:

> This restriction ensures that no unconstrained object is returned by a function.

No_Mixed_Positional_And_Named_Associations:

> This restriction ensures that there are nor mixed positional and named associations any where in the program.

No_Decimal_Fixed_Points:

> Until decimal fixed point and digit or delta constraint are allowed this restriction ensures at compile time that not decimal types is used. We can also use Ada RM H.4 restriction No_Fixed_Point.

No_Boolean_Subtypes:

This restriction ensures that there are no subtype of boolean types, considered to be an enumeration type.

No_Default_Expression_In_Record:

This restriction ensures that default initial values in record component cannot be given because the values appear remote from where they are used.

No_Aspect_Clause_In_Record:

This restriction ensures that there are no aspect clause in record.

No_Goto_Statements:

This restriction ensures that in compile time there are no goto statement.

No_Outer_Loop_In_Exit_Statement:

This restriction ensures that exit statements always apply to their very neighbor enclosing loops.

No_Multiple_Return_Statement:

This restriction ensures that return statements appear once and only once as a last statement in function subprogram. Theoretically this should be done in preprocessing phase, but this is not a priority for Customers.

2.4 Users requirements

We list below the uppermost priority extensions to be included in the Profile according to one AdaCore customer.

1. Overloaded subprograms (and operators?).
2. Automatic insertion of type qualifiers.
3. Anonymous types.
4. Derived scalar types.
5. Block statements.
6. Limitations on array slicing.
7. Limitations on array concatenation.
8. Restrictions on returns and exits.

Another customer based on its practice of SPARK frame suggests:

1. Attach an important priority to array slicing according to its experience, they slice frequently for performance reasons.
2. After that, probably automatic type qualifiers.
3. Then, derived integer types.
4. Finally, block statements.

2.4.1 Priority of extension in Sparkify tool

In order to provide quickly a prototype, after comparing Ada 2005 and SPARK syntax and after examing SPARK user's requirements; we classified by the different Ada constrictions we wish to translate into SPARK as below.

1. Generation of pragma asserts, preconditions and postconditions as check in SPARK syntax (–#).
2. Generation of global annotations: data-flow global analyse.
3. Typing: translating types qualifiers on aggregates, anonymous types and derived scalar types.
4. Naming: allowing overloading of subprograms and operators, and within in enumeration literals, allowing also renaming.
5. Expressions: Allowing array slicing/sliding and array concatenation.
6. Statement: forbidding goto statement, restricting on returns and exits statements and allowing blocks statements.
7. Handling of mutual dependencies and recursion.
8. Generic and formal type declarations.
9. Discriminant types.
10. Dynamic types.

2.5 Adopted approach

In the following chapter, to each extension we judged to be of uppermost priority and while remaining easy to implement within the Sparkify tool (already described in the section 1.4.4) is assigned a TN (AdaCore work flow Ticket Number). We also use Test Driven Development (TDD) simplified method. First, we describe briefly one of the extensions identified previously. Second, we implement the corresponding tests and perform a design corresponding to these tests. Finally, we refine the design in an attempt to do global test covering. Each design is given in the form of a commented pseudo-code that can be easily read by users and that can be used in any other future versions.

2.6 Conclusion

In this chapter, we have defined the characteristics of a subset of Ada, ideal for verification purposes in the context of current SPARK verification tools. After comparing Ada language and SPARK language in regard to Hi-Lite context, we have presented a canvas of a Verification Profile for Ada. Thus, we have summed up the possible extensions that may increase the expressive power of the Ada subset that can be verified via current SPARK tools. We have also summarized the Profile restrictions that is, the Ada entities that remain outside any translation policy, to maintain easiness of verification. Indeed, the Profile provides more and more expressive syntax by lifting some limitations of SPARK that do not make the verification intractable (allowing anonymous type, derived scalar type, qualified aggregate expression, ...). The implementation of the proposed Profile will make it possible to prove properties of Ada programs that do not respect some restrictions in SPARK with respect to Ada (e.g., absence of direct or mutual restriction, the absence of side-effects in functions, ...).

In the next chapter we explain the entire design and implementations of some extensions (based on the priority listed and the adopted approach) that we performed in Sparkify tool.

3 Technical design and implementation

3.1 Introduction

In this section we present some technical designs of the suggested translations implemented in the Sparkify tool. As we have already seen in section 1.4.4, Sparkify is implemented as various traversals of the sub-tree of each kind of node during the main traversal of the AST. During a traversal, a node can incur a special treatment either during in the pre-operation, i.e. when a node is treated before its children nodes are visited, or in the post-operation, i.e. when a node is treated after its children nodes have been visited. Thus, each design of an extension corresponds to a call-back used as a pre-operation or a post-operation for some kind of node.

3.2 Method of Development

As noticed previously in section 2.5, we used a simplified version of Test Driven Development (TDD). For each extension, we started by adding new test cases checking the correctness of the suggested extension. These tests were not initially run as part of the regression test-suite until the extention was implemented. Then we implemented the extension and checked both that if fixed the specific tests developed for this extension and that it did not create any regression on the existing test-suite. After that, we could add the specific tests to the regression test-suite and commit our modifications to the main branch of development. In the following, each design is given in the form of a commented pseudo-code.

3.3 Design of my Own implementation in Sparkify

Sparkify realized now many extensions that we propose in 2.3.3. Here I present my own implementation in Sparkify.

3.3.1 Extension_Of_Type_Qualifier_On_Aggregate

Description:

This is an extension suggested in section 2.3.2 in LRM rule 4.3. In fact, SPARK does not allow aggregates in expressions. Only aggregates that are prefixed with a type qualifier are allowed, which forces users to manually insert these. This extension consists in translating aggregates without a type qualifier in Ada into aggregates prefixed by the appropriate type qualifier in SPARK.

Tests:

Type_Qualifiers_1: record aggregate expression must be qualified with a subtype mark.

```
P := (mReal => 0, mImaginary => 1);
```

Type_Qualifiers_3: array aggregate expression must be qualified with a corresponding subtype.

```
Initial_Board := (1 => ((White,Rook), (White,Knight), (White,Bishop)),
                  2 => (Board_Index =>(White,Pawn)), 3 .. 6 => ... ));
```

Design:

The following is the suggested design (pseudo-code) implemented in the Sparkify tool (TN J416-001).

```
define An_Aggregate_Pre_Op as
-- to be called on any aggregate expression (Element)

   get enclosing element (Encl_Element) of Element

   if Encl_Element is of kind A_Qualified_Expression
   then
        do nothing -- because the aggregate is already qualified

   else
        get the declaration of the type of the aggregate (Type_Decl)

        if Type_Decl is null or is An_Unconstrained_Array_Definition
        -- This occurs when the name does not exist in the original
        -- source code.
        -- Then, we must have created it instead during the translation.
        then
            get type name (Type_Str) in current context
            -- Retrieve it from the map created during the translation.
        else
            get the unique name (Base_Name) of Type_Decl

            get the full name (Full_Name) of Base_Name
            -- In SPARK, identifiers should be completely qualified.

            get the image (Type_Str) of Full_Name

    echo source text from Echo_Cursor to before Element
    print Type_Str & "'"
    position Echo_Cursor on Element
    -- The following call to echo will start by printing Element.
```

3.3.2 Extension_Of_Anonymous_Types

Description:

This is an extension suggested in section 2.3.2 in LRM rules 3.3.1 and 3.5. In fact, SPARK does not allow anonymous types, all subtypes (types with constraints) should be named, while Ada allows them in many places: for both the type of index and the type of element in an array type; directly in a definition of a variable (for array, scalar type and constant); as the type of a component in a record definition. This extension consists in translating uses of anonymous types in Ada in uses of named types in SPARK, with a preceding definition for the new named types created.

Tests:

Anonymous_Type_1: Test anonymous type in object declarations or definitions.

```
Var_Out : Integer range 0 .. 10 := 0;
My_Array1 : array (1.. 10) of Integer;
Ten_Characters : String(1 .. 10);
```

Anonymous_Type_2: Test the anonymous type in type_declaration with constrained_array_definition

```
type Array2 is array (5 .. 9) of Integer range 0 .. 9;
```

Anonymous_Type_3: Test the anonymous type as component_list in the record type

```
type Date is record
        Day : Integer range 1 .. 31;
        Year : Integer range 1 .. 4000;
        Day_String : String(1 .. 6);
end record;
```

Anonymous_Type_4: automatic type insertion in for-loop (part of J512-016)

```
for I in 1 .. 9 loop
for J in Buffer'Range (2) loop
```

Anonymous_Type_5: test of anonymous array definition with initialization that must be qualified by the new subtype name.

```
type Tab is array(Integer range <>) of Integer range 0 .. 10;
T : Tab (0 .. 9);
T := (others => 0);
X : array (Integer range 0 .. 10) of Integer := (others => 0);
Y := T (1) + X (1);
```

Design:

The following is the suggested design (pseudo-code) implemented in the Sparkify tool (TN J423-017 and TN J512-016)

```
define A_Object_Declaration_Pre_Op as
-- to be called on any object or variable declaration (Element)

   get object definition view (Object_Def)
   -- The object view is the expression following the reserved word "is"
   -- in the object declaration.

   case the definition kind of Object_Def
       when A_Subtype_Indication or A_Type_Definition
       do
          -- Echo the code preceding the declaration in order to allow
          -- the definition of new types here.
          echo source text from Echo_Cursor to before Element
          position Echo_Cursor on Element

          case the definition kind of Object_Def
              when A_Subtype_Indication
              do
                 if the subtype indication is a simple subtype mark
                 then
                      return

                 else
                      get the subtype name (Subtype_Name) of Object_Def
                      -- by calling Transform_Subtype_Indication which:
                      -- . rejects the code if "non null" (for access types)
                      --   is present;
                      -- . prints a new subtype definition for the current
                      --   subtype, with name (Subtype_Name).

              when A_Type_Definition
              do
                 -- Object_Def is the type of a constrained array here

                 get the subtype string (Subtype_Str) of Object_Def
                 -- by calling Transform_Constrained_Array_Definition which:
                 -- . prints new subtype definitions for every anonymous
                 --   array index types;
                 -- . prints a new subtype definition for the anonymous
                 --   array component;
                 -- . returns the printed definition of the array subtype

                 create a new fresh name for the new subtype name (Subtype_Name)
                 print the new definition of Subtype_Str with name Subtype_Name

              when others
              do
                 internal ERROR

          echo source text from Echo_Cursor to before Object_Def
          -- to print the identifier(s) being defined

          if Element is A_Constant_Declaration
          -- semantically An object_declaration with the reserved word constant
          -- declares a constant object
          then
              print "constant␣" & Subtype_Name
          else
              print the new subtype name Subtype_Name

          position Echo_Cursor after Element
          -- The following call to echo will start by printing after Element.

          store the new name Subtype_Name by calling Store_New_Name
          --  Record that a new name was created for the declaration.

       when others
       do
          reject the code -- unexpected element
```

```
define A_Type_Declaration_Pre_Op as
-- to be called on any type declaration (Element)

   -- Echo the code preceding the declaration in order to allow
   -- the definition of new types here.
   echo source text from Echo_Cursor to before Element
   position Echo_Cursor on Element

   get type declaration view (Type_Def)
   -- The declaration view is a expression following the reserved word "is"
   -- in the type declaration.

   case the type kind of Type_Def
       when A_Constrained_Array_Definition
       do
          get the subtype string (Array_Type_Str) of Type_Def
          -- by calling Transform_Constrained_Array_Definition which:
          -- . prints new subtype definitions for every anonymous
          --   array index types;
          -- . prints a new subtype definition for the anonymous
          --   array component;
          -- . returns the printed definition of the array subtype

          echo source text from Echo_Cursor to before Type_Def
          -- to print the identifier being defined

          print Array_Type_Str & ";"
          position Echo_Cursor after Element

       when An_Unconstrained_Array_Definition
       do
          -- Only the type of array component needs to be treated.

          get the components of array (Array_Comp_Def)
          get the component definition view (Comp_Def) of Array_Comp_Def
          -- this is a subtype indication

          case the definition kind of Comp_Def
              when A_Subtype_Indication
              do
                 echo source text from Echo_Cursor to before Comp_Def
                 -- to print until "of"

                 print the new subtype name (Subtype_Name) of Comp_Def
                 -- by calling Transform_Subtype_Indication which:
                 -- . rejects the code if "non null" (for access types)
                 --   is present;
```

```
                -- . prints a new subtype definition for the current
                --   subtype, with name (Subtype_Name).

                echo ";"
                position Echo_Cursor after Element

             when others
             do
                reject the code -- unexpected element

         when A_Record_Type_Definition
         do
            -- Handle anonymous types in record component definition.

            get the record definition (Record_Def) of Type_Def

            if a definition kind of Record_Def is A_Null_Record_Definition
            then
                if flat element kind of Record_Def is A_Tagged_Record_Type_Definition
                then
                    return
                    -- In the Profile, as in SPARK, a record definition
                    -- cannot be a null record unless it is tagged.
                else
                    internal ERROR on untagged null record definition

            else
                get the list record components (Record_Comps) of Record_Def

                if Record_Comps is of length 1 and it is a A_Null_Component
                then
                    if flat element kind of Record_Def is
                      A_Tagged_Record_Type_Definition
                    then
                        return
                        -- In the Profile, as in SPARK, a component list
                        -- cannot be null unless the record is tagged.
                    else
                        internal ERROR on untagged null component in record

                -- Print new subtypes for each anonymous subtype used in
                -- in the type of a component.
                foreach Record_Comps (J)
                do
                   get the subtype indication of each component view (Object_Def)
                   get the new names created in array (Subtype_Names)
                   print a new subtype
                   -- by calling Transform_Subtype_Indication as previously

                echo from Echo_Cursor to before the first component

                foreach Record_Comps (J)
                do
                   echo from Record_Comps(J) to before the corresponding
                     Object_Def
                   -- to print the identifier(s) being defined
                   print the new Subtype_Names (J) previously collected
                   print ";"

                print "end␣record;"
                position Echo_Cursor to after Element

         when others
         do
            nothing
```

3.3.3 Extension_Of_Derived_Scalar_Types

Description:

The following is an extension suggested in section 2.3.2 in LRM rule 3.4. SPARK does not allow derived types for scalar types. Only record types extensions are allowed. Sparkify translates these into subtypes in SPARK.

Tests:

The first solution that we implemented is based on the proposition of the SPARK roadmap to translate a derived type into a new type by asserting the base type like
– from:
type Counter **is new** Positive;
type Index_Int **is new** Integer **range** 1 .. 10;
– to:
type Counter **is** Positive'First .. Positive'Last;
type Index_Int **is range** 1 .. 10;
–# assert Index_Int'Base is Integer;

But this solution only works for integer types, not enumeration types. Thus, we proposed another solution which consists in translating a derived type in Ada into a subtype in SPARK.
– from:
type Counter **is new** Positive;
type Midweek **is new** Day **range** Tue .. Thu;
– into:
subtype Counter **is** Positive;
subtype Midweek **is** Day **range** Tue .. Thu;

Design:

The following is the suggested design (pseudo-code) implemented in Sparkify tool (TN J630-013)

```
define A_Derived_Type_Definition_Pre_Op as
-- to be called on any derived type definition (Element)

   get the parent subtype indication (Subtype_Ind)
   -- by calling Parent_Subtype_Indication

   get the name (Encl_Decl_Name) of the enclosing element

   echo source text from Echo_Cursor to before Element

   print "subtype" &  Encl_Decl_Name & "is"

   position Echo_Cursor on Subtype_Ind
   -- The following call to echo will start by printing the subtype.
```

3.3.4 Extension_Of_Subprograms_Overloading

Description:

The following is an extension suggested in section 2.3.2 in LRM rule 6.1. In fact, SPARK does not allow the overloading of subprograms (both functions and procedures), nor overloading of operators, nor overloading of enumeration literals. In Sparkify, it is just a matter of renaming the subprograms or operators to discriminate between overloaded ones. For instance, to detect which subprograms are overloaded, the easiest way is to maintain a set of names of subprograms in scope, which is created afresh for each new unit, and augmented with all the subprograms of a declarative block each time a declarative block is entered. For operators it is necessary to use the prefix notation with the new names and the parameters will be the operands. By contrast, however, overloading of enumeration literals is more complex and is still to be done.

Tests:

```
procedure Swap (X, Y : in out Integer);
procedure Swap (X, Y : in out Float);

procedure Exchange (A, B : out Integer; C, D : out Float; E, F : out Long_Integer) is
   procedure Swap (X, Y : in out Long_Integer);
   procedure Swap (X, Y : in out Long_Integer) is
      T : Long_Integer;
   begin
      T := X; X:= Y; Y:= T;
   end Swap;
begin
   A := 10; B := 20; C := 10.0; D := 20.0; E := 100000; F := 200000;
   Swap (A,B);
   Swap (C,D);
   Swap (E,F);
end Exchange;
```

Design:

The following is the suggested design (pseudo-code) implemented in Sparkify tool (TN J622-019)

```
define Return_Overloaded_Name as
-- to be called on any declaration element (Decl)
-- to detect if there are overloading in :
-- 1- subprograms
-- 2- operators (not implemented yet)
-- 3- enumeration literals (not implemented yet)
-- In these cases, it returns a new name. Otherwise, it returns the name.

    get the defining name (Def_Name) defined by Decl

    get the new name (New_Name) registered previously in the map
    -- by calling Get_New_Name which uses the source location of
    -- the declaration to identify it

    if New_Name is not empty
    then
        -- the Def_Name is located and is in the map
        return New_Name
    else
        if Def_Name is in the set of overloaded names (Nameset)
        then
            create a fresh name (Fresh_Name) for Def_Name
            store Fresh_Name in the map
            -- by calling Store_New_Name based on the location of
            -- the declaration

            return Fresh_Name
        else
            insert Def_Name in Nameset
            store Def_Name in the map
            return Def_Name


call Return_Overloaded_Name in the right places
-- In the right call-back (A_Subprogram_Unit_Declaration_Pre_Op,
-- A_Subprogram_Unit_Pre_Op, An_Identifier_Pre_Op) we maintain a set
-- of names of subprograms in scope, which is created a fresh (returned
-- by Return_Overloaded_Name) for each new unit, and augmented with all
-- the subprograms of a declarative block each time a declarative block
-- is entered.
```

3.4 Conclusion

In this section we have presented the technical design corresponding to some of our proposed translation of the larger subset of Ada into SPARK. Some of the transformations we realized in the Sparkify tool show how some of the limitations of SPARK with respect to Ada (anonymous type, derived scalar type, qualified aggregate expression, ...) merely amounts to programming style rules imposed on users to increase the safety of programs, without having a real impact on the tractability of verification. With another extension not presented here, we also checked that it is possible to prove properties of Ada programs that do not respect some more fundamental restrictions in SPARK (e.g., absence of direct or mutual recursion). Finally, some properties like the absence of side-effects in functions and certain Ada features (dynamic dispatching, exception, generics, ...) really need a real extension inside SPARK language.

In the following chapter we present the environment that we used during our work and to collaborate with other people on the project.

4 Realization

4.1 Introduction

In this chapter we summarize the work we realized during the internship and the environment that allowed us to complete this work.

4.2 Equipments Environment

In this work, I used:

▷ A Dell laptop for our local development

From the software perspective, I have worked on a Linux platform, which includes the tools needed to make this work:

▷ Linux Operating System (Ubuntu)

▷ GNAT Pro for Ada programming (include GPS IDE)

▷ SPARK tool set {Examiner (spark), Simplify (sparksimp), POGS (pogs)} for SPARK programming.

▷ ASIS-for-GNAT : API to process compilable sources (AST).

▷ Sparkify : Translator from Ada sources to SPARK for verification purpose, based on ASIS-for-GNAT

▷ Git repository : Git is a free & open source, distributed version control system used in Hi-Lite project (referenced in Open-DO Forge) [7]. Briefly, I define here the simple work flow I used. Like all other developers, I have my local cloned repository for my local developments which mirrors a personal remote repository on the public forge. When my local developments are satisfying, I push them in my remote repository, from which my supervisor pulls my changes and pushes them in the common repository on the public forge, with possible modifications.

4.3 Contribution and profit

During this internship I learned a lot.

Having started my training in February, I participated in the majority of the meetings and discussion of the Hi-Lite project, which was launched on May 4 [8]. During these discussions I learned all of the upstream project formulations by the different project's partners : customers requirements and specifications. The universe of the project allowed me to explore the process of research and developement in industry and to discover applications of formal methods in industry in particular.

At the theoretical level, I have improved my knowledge of the Ada language and picked up the SPARK language which have enabled me to define the suggested Verification Profile for Ada. I have realized a comparison face-to-face and SPARK Ada BNF, presented in Annex A. This comparison should be useful for programmer of Ada and SPARK. For instance basing on this comparison, on Ada LRM [6] and on SPARK LRM [5] I have written a canvas for a Verification Profile for Ada (the technical document presented in chapter 2). This document is one of the references of the redaction of ALFA: Annotated Language for Functions in Ada; "ALFA is a sub-language of Ada 2012, the next version of the Ada standard, which identies which subprograms are t for formal verication".

In practical terms I have participated in the implementation of some extensions in the prototyping phase by using ASIS API, for which I have benefited from the good programming skills of the senior developers of Sparkify.

For sure, this progression has not been without difficulties, but difficulties were quickly relieved by the competence of people who helped me with goods suggestions.

As part of my assessment, this project has been a positive contribution to my technical, theoretical and professional skills in a dynamic and friendly environment.

4.4 Perspectives

In the context of both the Hi-Lite project and the SPARK toolset, there are two main ways of verifying an Ada program.

1. Using the rigorous SPARK programming language to benefit from its powerful verification support tools. In this case, verification tasks go through the following process:

   (a) Verification process:

   ```
   Input: SPARK program A to be analysed
   Use SPARK toolset
   Output: SPARK program A is verified or not
   ```

   (b) Advantage: Benefit from robust tools.

   (c) Drawback: The SPARK language is quite restrictive and impose the rigid programming style.

2. Define a verification Profile for Ada larger than SPARK relaxing some of the rigorous restrictions of SPARK and while being still able to benefit from SPARK verification tools. Two approaches can be followed in this case:

   ▷ The first solution is to extend SPARK by automatic translation while excluding all Ada features which may turn intractable the verification process.

(a) Verification process:

```
Input: Ada program A in the Profile
Translate A into SPARK A′
Use SPARK verification tools on A′
Output: Ada program A is verified or not
```

(b) Advantage: the Profile is more expressive than SPARK, and thereby, program A can be more expressive than before.

(c) Drawback: there exists more programs than before that cannot be automatically proved.

▷ The second solution is to automatically extract portions of the code respecting the verification profile.

(a) Verification process:

```
Input: Ada program A
for each entity I in A
  if I is in the Profile
  then
    Translate I into SPARK I′
    Use SPARK verification tools on I′
    Output: Ada entity I of program A is verified or not
  else
    Output: Ada entity I is not verified
```

(b) Advantage: allow formally proving all the code portions respecting the Profile, while the verification on the remaining parts can be undertaken using testing techniques.

(c) Drawback: Formal verification is only partially applied to the program.

The diagram below shows a possible place of the Profile in Hi-Lite vision, which is to integrate formal verification techniques and testing techniques.

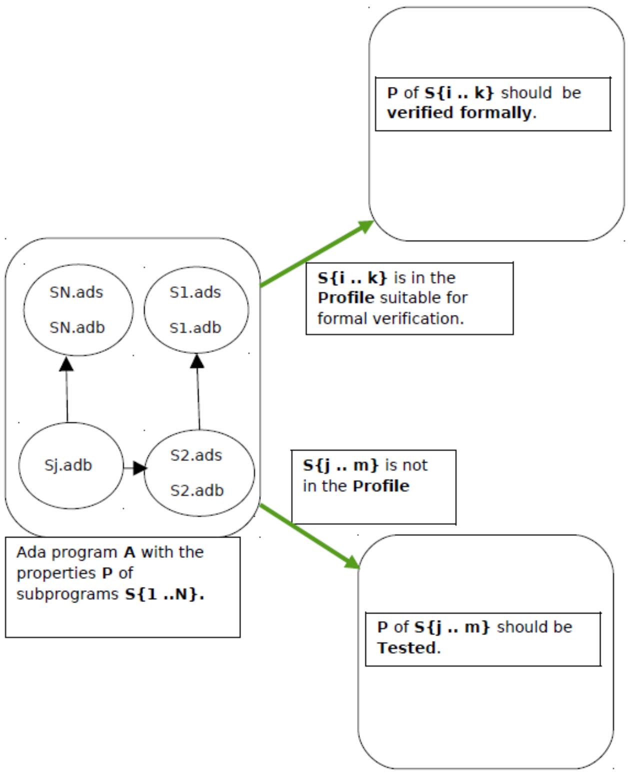
P of S{i .. k} should be verified formally.
SN.ads
SN.adb
S1.ads
S1.adb
S{i .. k} is in the Profile suitable for formal verification.
Sj.adb
S2.ads
S2.adb
S{j .. m} is not in the Profile
Ada program A with the properties P of subprograms S{1 ..N}.
P of S{j .. m} should be Tested.

FIG. 4.1 – Possible place of the Profile in Hi-Lite

4.5 Deadline

Tasks	Avril	May	June	July	August	September
Documentation	**X**	**X**				
BNF Comparison	**X**	**X**	**X**			
Design		**X**	**X**	**X**	**X**	
Implementation		**X**	**X**	**X**	**X**	
Tests & debug		**X**	**X**	**X**	**X**	**X**
Report redaction		**X**	**X**	**X**	**X**	**X**

TAB. 4.1 – Tasks Chronogram

4.6 Conclusion

In this chapter we have presented the work done during those six months of internship. Some features proposed for the verification profile have already been implemented in the prototype tool which, which shows that some limitations of SPARK can be lifted and that we can define a more expressive verification profile for Ada than SPARK. We have proposed some designs of the other features. However the different designs should be refined correctly in the final tool.

General Conclusion

Ada language, as certain high level languages, is a power language which provides interesting features to programmers. Besides, Ada has some very desirable features for critical software development. In addition, some features announced in Ada 2012 will make it even more suitable for safety-critical development and verification. However, its complete form is not suitable for formal verification because this requires a logical and coherent informal definition of the language (in particular non-determinism) or even a complete formal definition. On the contrary, SPARK uses a strict and rigorous subset of the Ada language that allow formal verification supported by industrial-strength tools.

In the context of Hi-Lite project, we have defined a Verification Profile for Ada based on SPARK, by identifying a subset of Ada that can be converted to SPARK. The Profile is also based on some restrictions of Ada which would make formal verification intractable, for which we propose new restrictions in the Profile (No_Access_Types, No_Aliasing, No_Exception_Handlers, ...).

On the one hand, this conversion provides syntactic sugar by lifting some limitations of SPARK which do not compromise automatic formal verification (we allowed anonymous types, derived scalar types, qualified aggregate expressions, ...). Some features of Ada (generic units, raising exceptions, dynamic storage allocation, ...) which are disallowed by SPARK due to their complicate formal definition should be also allowed with SPARK roadmap. On the contrary, access types (pointers) manipulations, which can involve aliasing, are extremely difficult to handle in formal verification so they should not be used in the Profile. However, others features which require dynamic storage allocation such as dynamically constrained arrays, discriminants and recursion can be allowed. Recursion is already operational in our prototyping tool.

On the other hand, this proposal Profile will not make it now possible to prove properties of Ada programs that do not respect some restrictions in SPARK (e.g. absence of side-effects in functions, loop invariants, ...).

During my internship many features of the suggested Profile have been implemented in the prototyping tool (Sparkify) and theirs designs should be used to refine the final solution. However everything is not yet done, so this document does not attempt to provide the full design of the final Verification Profile for Ada. It merely lists the possibles activities one might pursue to program with a subset of Ada for verification purpose.

This experience has been rewarding for us on the professional, theoretical and practical domain. Finally, we hope that this work provides a solid canvas on which we can weave an elegant embroidery.

Bibliographie

[1] Yannick Moy-Romain Berrendonner, High Integrity Lint Integrated with Testing and Execution, AdaCore (27 Nov 2009).

[2] Rod Chapman (SPARK Products manager), SPARK Pro roadmap, SPARK Pro Steering Committee, 19th August 2009.

[3] John Barnes. Programming in Ada 2005 . Addition-Wesley (2006).

[4] John Barnes with Praxis Critical Systems. High integrity Software - The Spark approach to Safety and Security. Addition-Wesley (2003).

[5] SPARK - The SPADE AdaKernel (including RavenSPARK) Edition 5.7.

[6] http://www.adaic.org/standards/05rm/html/RM-TOC.html

[7] http://forge.open-do.org/projects/hi-lite/

[8] http://informatique.annuairecommuniques.com/2010/05/adacore-presente-le-projet-hi-lite.html

A
Ada and SPARK's BNF comparison

This appendix summarizes and compares the full syntax of ADA language and SPARK language, taken respectively from Ada Reference Manual for Ada 2005 R2 [6], and SPARK2005 [5]. These two syntaxes are represented in BNF form.

As it is done in SPARK LRM, collected syntax of SPARK rules marked with an asterisk (*) are variants of the same rules of standard Ada and those marked with a plus (+) are additional rules.

Syntactic categories are represented by lower case names; some of these contain embedded underlines to increase readability.

Boldface words are used to denote reserved words, for example: **array**

If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example *subtype*_name and *task*_name are both equivalent to name alone.

We used Ada syntax tri-colored to make the syntactic distinction (for semantic difference see the description of the Profile in 2.3.1, for more information see the RM)

Red : set of Ada syntax that we will not be used in Verification Profile for Ada.

Green : set of Ada syntax that we added in the Profile.

Blue : set of Ada syntax which could be in the Profile or set of Ada syntax already in SPARK Pro Roadmap.

Summary of Ada and SPARK BNFs comparison

ADA	**SPARK**	**Notes**
	2.1: character ::= graphic_character | format_effector | other_control_function 2.1: graphic_character ::= identifier_letter | digit | space_character | special_character	Minor lexical difference. Ada used the Universal Character Set (ISO/IEC 10646:2003) whereas SPARK use only the full Latin-1 set.
2.3: identifier ::= identifier_start {identifier_start | identifier_extend} 2.3: identifier_start ::= letter_uppercase | letter_lowercase | letter_titlecase | letter_modifier | letter_other | number_letter 2.3: identifier_extend ::= mark_non_spacing | mark_spacing_combining | number_decimal | punctuation_connector | other_format 2.4: numeric_literal ::= decimal_literal | based_literal 2.4.1: decimal_literal ::= numeral [.numeral] [exponent]	2.3 identifier ::= identifier_letter { [underline] letter_or_digit } 2.3 letter_or_digit ::= identifier_letter | digit 2.4: numeric_literal ::= decimal_literal | based_literal 2.4.1: decimal_literal ::= numeral [.numeral] [exponent]	No major difference. In SPARK the identifiers predefined in its standard package version may not be redeclared.

2.4.1: numeral ::= digit {[underline] digit} 2.4.1: exponent ::= **E** [+] numeral | **E** – numeral 2.4.1: digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	2.4.1: numeral ::= digit {[underline] digit} 2.4.1: exponent ::= **E** [+] numeral | **E** – numeral	No difference. just digit is considered to be a basic terminal in SPARK
2.4.2: based_literal ::= base # based_numeral [.based_numeral] # [exponent] 2.4.2: base ::= numeral 2.4.2: based_numeral ::= extended_digit {[underline] extended_digit} 2.4.2: extended_digit ::= digit | A | B | C | D | E | F	2.4.2: * based_literal ::= base # based_numeral # [exponent] 2.4.2: base ::= numeral 2.4.2: based_numeral ::= extended_digit {[underline] extended_digit} 2.4.2: extended_digit ::= digit | A | B | C | D | E | F	Lexical difference. In SPAR, real literals may not be expressed in base form e.g 16#F.FF#E+2 .
2.5: character_literal ::= 'graphic_character'	2.5: character_literal ::= 'graphic_character'	
2.6: string_literal ::= "{string_element}" 2.6: string_element ::= "" | *non_quotation_mark*_graphic_character	2.6: string_literal ::= "{string_element}" 2.6: string_element ::= "" | *non_quotation_mark*_graphic_character	
2.7: comment ::= –{*non_end_of_line*_character}	2.7: comment ::= –{*non_end_of_line*_character}	
2.8: pragma ::= **pragma** identifier [(pragma_argument_association {,pragma_argument_association})]; 2.8: pragma_argument_association ::= [*pragma_argument*_identifier =>] name | [*pragma_argument*_identifier =>] expression	2.8: pragma ::= **pragma** identifier [(pragma_argument_association {,pragma_argument_association})]; 2.8: pragma_argument_association ::= [*pragma_argument*_identifier =>] name | [*pragma_argument*_identifier =>] expression	

3.1: basic_declaration ::= type_declaration | subtype_declaration | object_declaration | number_declaration | subprogram_declaration | abstract_subprogram_declaration | null_procedure_declaration | package_declaration | renaming_declaration | exception_declaration | generic_declaration | generic_instantiation 3.1: defining_identifier ::= identifier	3.1: * basic_declaration ::= type_declaration | subtype_declaration | object_declaration | number_declaration 3.1: defining_identifier ::= identifier	In SPARK for subprogram declarations see 7.1 and Annex B.1; for package declarations see 3.11; and for renaming declarations see 7.1 and 3.11; because they are not considered to be basic declarations in SPARK.
3.2.1: type_declaration ::= full_type_declaration | incomplete_type_declaration | private_type_declaration | private_extension_declaration 3.2.1: full_type_declaration ::= **type** defining_identifier [known_discriminant_part] **is** type_definition; | task_type_declaration | protected_type_declaration 3.2.1: type_definition ::= enumeration_type_definition | integer_type_definition | real_type_definition | array_type_definition | record_type_definition | access_type_definition | derived_type_definition | interface_type_definition	3.2.1: * type_declaration ::= full_type_declaration | private_type_declaration | private_extension_declaration 3.2.1: * full_type_declaration ::= **type** *defining*_identifier **is** type_definition; | task_type_declaration | protected_type_declaration 3.2.1: * type_definition ::= enumeration_type_definition | integer_type_definition | real_type_definition | array_type_definition | record_type_definition | modular_type_definition | record_type_extension 3.2.1: + record_type_extension ::= **new** type_mark **with** record_definition ;	

3.2.2: subtype_declaration ::= **subtype** defining_identifier **is** subtype_indication; 3.2.2: subtype_indication ::= [null_exclusion] subtype_mark [constraint] 3.2.2: subtype_mark ::= *subtype*_name 3.2.2: constraint ::= scalar_constraint | composite_constraint 3.2.2: scalar_constraint ::= range_constraint | digits_constraint | delta_constraint 3.2.2: composite_constraint ::= index_constraint | discriminant_constraint	3.2.2: subtype_declaration ::= **subtype** defining_identifier **is** subtype_indication; 3.2.2: subtype_indication ::= subtype_mark [constraint] 3.2.2: subtype_mark ::= *subtype*_name 3.2.2: constraint ::= scalar_constraint | composite_constraint 3.2.2: * scalar_constraint ::= range_constraint 3.2.2: * composite_constraint ::= index_constraint | discriminant_constraint	Lack of an asterix (*) before the SPARK rule 3.2.2: subtype indication Remove the additional * before composite constraint because they are identical.
3.3.1: object_declaration ::= defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]; | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression]; | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]; | single_task_declaration | single_protected_declaration 3.3.1: defining_identifier_list ::= defining_identifier {,defining_identifier}	3.3.1: * object_declaration ::= defining_identifier_list : [**aliased**] [**constant**] subtype_mark [:= expression]; 3.3.1: defining_identifier_list ::= defining_identifier {,defining_identifier}	

3.3.2: number_declaration ::= defining_identifier_list : **constant** := *static*_expression;	3.3.2: number_declaration ::= defining_identifier_list : **constant** := *static*_expression;	
3.4: derived_type_definition ::= [**abstract**] [**limited**] **new** *parent*_subtype_indication [[**and** interface_list] record_extension_part]	3.4: *	In SPARK, record extension is already defined in 3.2.1.
3.5: range_constraint ::= **range** range 3.5: range ::= range_attribute_reference | simple_expression .. simple_expression	3.5: * range_constraint ::= **range** *static*_range 3.5: range ::= range_attribute_reference | simple_expression .. simple_expression	In SPARK, the range in a range constraint shall be static, no static range shall be a null range
3.5.1: enumeration_type_definition ::= (enumeration_literal_specification {, enumeration_literal_specification}) 3.5.1: enumeration_literal_specification ::= defining_identifier | defining_character_literal 3.5.1: defining_character_literal ::= character_literal	3.5.1: enumeration_type_definition ::= (enumeration_literal_specification {,enumeration_literal_specification}) 3.5.1: * enumeration_literal_specification ::= *defining*_identifier	Since the character literals belong to the enumeration type Character in package Standard, character literals cannot be used as enumeration literals
3.5.4: integer_type_definition ::= signed_integer_type_definition | modular_type_definition 3.5.4: signed_integer_type_definition ::= **range** *static*_simple_expression .. *static*_simple_expression 3.5.4: modular_type_definition ::= **mod** *static*_expression	3.5.4: * integer_type_definition ::= signed_integer_type_definition 3.5.4: signed_integer_type_definition ::= **range** *static*_simple_expression .. *static*_simple_expression 3.5.4: modular_type_definition ::= **mod** *static*_expression	In SPARK modular type definition is in 3.2.1
3.5.6: real_type_definition ::= floating_point_definition | fixed_point_definition	3.5.6: real_type_definition ::= floating_point_definition | fixed_point_definition	

3.5.7: floating_point_definition ::= **digits** *static*_expression [real_range_specification] 3.5.7: real_range_specification ::= **range** *static*_simple_expression .. *static*_simple_expression	3.5.7: * floating_point_definition ::= **digits** *static*_simple_expression [real_range_specification] 3.5.7: real_range_specification ::= **range** *static*_simple_expression .. *static*_simple_expression	
3.5.9: fixed_point_definition ::= ordinary_fixed_point_definition | decimal_fixed_point_definition 3.5.9: ordinary_fixed_point_definition ::= **delta** *static*_expression real_range_specification 3.5.9: decimal_fixed_point_definition ::= **delta** *static*_expression **digits** *static*_expression [real_range_specification] 3.5.9: digits_constraint ::= **digits** *static*_expression [range_constraint]	3.5.9: * fixed_point_definition ::= ordinary_fixed_point_definition 3.5.9: * ordinary_fixed_point_definition ::= **delta** *static*_simple_expression real_range_specification 3.5.9: *	SPARK does not have decimal fixed point types and hence has no decimal fixed point definitions.
3.6: array_type_definition ::= unconstrained_array_definition | constrained_array_definition 3.6: unconstrained_array_definition ::= **array** (index_subtype_definition {, index_subtype_definition}) **of** component_definition 3.6: index_subtype_definition ::= subtype_mark **range** <> 3.6: constrained_array_definition ::= **array** (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition	3.6: array_type_definition ::= unconstrained_array_definition | constrained_array_definition 3.6: unconstrained_array_definition ::= **array** (index_subtype_definition {, index_subtype_definition}) **of** component_definition 3.6: index_subtype_definition ::= subtype_mark **range** <> 3.6: constrained_array_definition ::= **array** (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition	

3.6: discrete_subtype_definition ::= *discrete*_subtype_indication | range 3.6: component_definition ::= **[aliased]** subtype_indication | **[aliased]** access_definition	3.6: * discrete_subtype_definition ::= *discrete*_subtype_mark 3.6: * component_definition ::= subtype_mark	
3.6.1: index_constraint ::= (discrete_range {, discrete_range}) 3.6.1: discrete_range ::= *discrete*_subtype_indication | range	3.6.1: * index_constraint ::= (*discrete*_subtype_mark {, *discrete*_subtype_mark }) 3.6.1: * discrete_range ::= discrete_subtype_indication | *static*_range	Concern the static range and anonymous type
3.7: discriminant_part ::= unknown_discriminant_part | known_discriminant_part 3.7: unknown_discriminant_part ::= (<>) 3.7: known_discriminant_part ::= (discriminant_specification {; discriminant_specification}) 3.7: discriminant_specification ::= defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression] | defining_identifier_list : access_definition [:= default_expression] 3.7: default_expression ::= expression	3.7 *	In SPARK, the only forms of discriminants supported are in task type (see 9.1 for Ravenscar Profile) and in protected type discriminants (see 9.4) .
3.7.1: discriminant_constraint ::= (discriminant_association {, discriminant_association}) 3.7.1: discriminant_association ::= [discriminant_selector_name {| discriminant_selector_name} =>] expression	3.7.1 *	

3.8: record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition 3.8: record_definition ::= **record** component_list **end record** | **null record** 3.8: component_list ::= component_item {component_item} | {component_item} variant_part | **null**; 3.8: component_item ::= component_declaration | aspect_clause 3.8: component_declaration ::= defining_identifier_list : component_definition [:= default_expression];	3.8: * record_type_definition ::= [**tagged**] record_definition 3.8: * record_definition ::= **record** component_list **end record** | **null record** 3.8 * component_list ::= component_item { component_item} | **null** 3.8: * component_item ::= component_declaration 3.8: * component_declaration ::= defining_identifier_list : component_definition;	A record definition cannot be null record unless it is tagged. Note that a tagged null record serves only as a basis for type extension; direct use of the null record is not possible. In SPARK, aspect clause is renamed representation clause (see 13.1).
3.8.1: variant_part ::= **case** discriminant_direct_name **is** variant {variant} **end case**; 3.8.1: variant ::= **when** discrete_choice_list => component_list 3.8.1: discrete_choice_list ::= discrete_choice {| discrete_choice} 3.8.1: discrete_choice ::= expression | discrete_range | **others**	3.8.1: * 3.8.1: discrete_choice_list ::= discrete_choice {| discrete_choice} 3.8.1: * discrete_choice ::= *static*_simple_expression | discrete_range	
3.9.1: record_extension_part ::= **with** record_definition	3.9.1: *	In SPARK, already defined in 3.2.1.
3.9.3: abstract_subprogram_declaration ::= [overriding_indicator] subprogram_specification **is abstract**;		

3.9.4: interface_type_definition ::= [**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list] 3.9.4: interface_list ::= *interface*_subtype_mark {**and** *interface*_subtype_mark}		
3.10: access_type_definition ::= [null_exclusion] access_to_object_definition | [null_exclusion] access_to_subprogram_definition 3.10: access_to_object_definition ::= **access** [general_access_modifier] subtype_indication 3.10: general_access_modifier ::= **all** | **constant** 3.10: access_to_subprogram_definition ::= **access** [**protected**] **procedure** parameter_profile | **access** [**protected**] **function** parameter_and_result_profile 3.10: null_exclusion ::= **not null** 3.10: access_definition ::= [null_exclusion] **access** [**constant**] subtype_mark | [null_exclusion] **access** [**protected**] **procedure** parameter_profile | [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile	3.10: *	
3.10.1: incomplete_type_declaration ::= **type** defining_identifier [discriminant_part] [**is tagged**];	3.10.1: *	

3.11: declarative_part ::= {declarative_item}	3.11 * declarative_part ::= { renaming_declaration } { declarative_item | embedded_package_declaration | external_subprogram_declaration }	In SPARK, there are restrictions on the position of renaming declarations (which are not basic declarations in SPARK) (see 3.1).
3.11: declarative_item ::= basic_declarative_item | body	3.11: * declarative_item ::= basic_declarative_item | body | generic_function_instatiation	
3.11: basic_declarative_item ::= basic_declaration | aspect_clause | use_clause	3.11: * basic_declarative_item ::= basic_declaration | representation_clause	See generic function instatiation description in 12.3.
	3.11 + embedded_package_declaration ::= package_declaration { renaming_declaration | use_type_clause }	
	3.11 + external_subprogram_declaration ::= subprogram_declaration **pragma** Import (pragma_argument_association, pragma_argument_association { , pragma_argument_association });	
3.11: body ::= proper_body | body_stub	3.11: body ::= proper_body | body_stub	
3.11: proper_body ::= subprogram_body | package_body | task_body | protected_body	3.11: * proper_body ::= subprogram_body | package_body	

4.1: name ::= direct_name | explicit_dereference | indexed_component | slice | selected_component | attribute_reference | type_conversion | function_call | character_literal 4.1: direct_name ::= identifier | operator_symbol 4.1: prefix ::= name | implicit_dereference 4.1: explicit_dereference ::= name.**all** 4.1: implicit_dereference ::= name	4.1: * name ::= direct_name | indexed_component | selected_component | attribute_reference | function_call 4.1: * direct_name ::= identifier 4.1: * prefix ::= name 4.1: *	character literals, operator symbols and type conversions are not names, they are the primary (see 4.4)
4.1.1: indexed_component ::= prefix(expression {, expression})	4.1.1: indexed_component ::= prefix(expression {, expression})	
4.1.2: slice ::= prefix(discrete_range)	4.1.2: *	
4.1.3: selected_component ::= prefix.selector_name 4.1.3: selector_name ::= identifier | character_literal | operator_symbol	4.1.3: selected_component ::= prefix.selector_name 4.1.3: * selector_name ::= identifier	In SPARK a selector name cannot be a charac-ter_literal or an opera-tor_symbol.

4.1.4: attribute_reference ::= prefix'attribute_designator 4.1.4: attribute_designator ::= identifier[(*static*_expression)] | Access | Delta | Digits 4.1.4: range_attribute_reference ::= prefix'range_attribute_designator 4.1.4: range_attribute_designator ::= **Range** [(*static*_expression)]	4.1.4: attribute_reference ::= prefix'attribute_designator 4.1.4: * attribute_designator ::= identifier [(expression [, expression])] | **Delta** | **Digits** 4.1.4: range_attribute_reference ::= prefix´range_attribute_designator 4.1.4: range_attribute_designator ::= **Range** [(static_expression)]	
4.3: aggregate ::= record_aggregate | extension_aggregate | array_aggregate	4.3:	Minor missing of *
4.3.1: record_aggregate ::= (record_component_association_list) 4.3.1: record_component_association_list ::= record_component_association {, record_component_association} | **null record** 4.3.1: record_component_association ::= [component_choice_list =>] expression | component_choice_list => <> 4.3.1: component_choice_list ::= component_selector_name {| component_selector_name} | **others**	4.3.1: * record_ aggregate ::= positional_record_aggregate | named_record_aggregate 4.3.1: + positional_record_aggregate ::= (expression { , expression }) 4.3.1: + named_record_aggregate ::= (record_component_association{ , record_component_association }) 4.3.1: * record_component_association ::= component_selector_name => expression 4.3.1: *	

4.3.2: extension_aggregate ::= (ancestor_part **with** record_component_association_list) 4.3.2: ancestor_part ::= expression | subtype_mark	4.3.2: * extension_aggregate ::= (ancestor_part **with** record_component_association_list) | (ancestor_part **with null record**) 4.3.2: * ancestor_part ::= expression 4.3.2: +record_component_association_list ::= named_record_component_association | posi- tional_record_component_association 4.3.2: +positional_record_component_association ::= expression { , expression } 4.3.2: +named_record_component_association ::= record_component_association { , record_component_association }	
4.3.3: array_aggregate ::= positional_array_aggregate | named_array_aggregate 4.3.3: positional_array_aggregate ::= (expression, expression {, expression}) | (expression {, expression}, **others** => expression) | (expression {, expression}, **others** => <>) 4.3.3: named_array_aggregate ::= (array_component_association {, array_component_association}) 4.3.3: array_component_association ::= discrete_choice_list => expression | discrete_choice_list => <>	4.3.3: array_aggregate ::= positional_array_aggregate | named_array_aggregate 4.3.3: * positional_array_aggregate ::= (aggregate_item , aggregate_item { , aggregate_item }) | (aggregate_item {, aggregate_item} , **others** => aggregate_item) 4.3.3: * named_array_aggregate ::= (array_component_association {, array_component_association } [, **others** => aggregate_item]) | (**others** => aggregate_item) 4.3.3: + aggregate_item ::= expression | array_aggregate	

4.4: expression ::= relation {**and** relation} | relation {**and then** relation} | relation {**or** relation} | relation {**or else** relation} | relation {**xor** relation} 4.4: relation ::= simple_expression [relational_operator simple_expression] | simple_expression [**not**] **in** range | simple_expression [**not**] **in** subtype_mark 4.4: simple_expression ::= [unary_adding_operator] term {binary_adding_operator term} 4.4: term ::= factor {multiplying_operator factor} 4.4: factor ::= primary [** primary] | **abs** primary | **not** primary 4.4: primary ::= numeric_literal | **null** | string_literal | aggregate | name | qualified_expression | allocator | (expression)	4.4: expression ::= relation {**and** relation} | relation {**and then** relation} | relation {**or** relation} | relation {**or else** relation} | relation {**xor** relation} 4.4: relation ::= simple_expression [relational_operator simple_expression] | simple_expression [**not**] **in** range | simple_expression [**not**] **in** subtype_mark 4.4: simple_expression ::= [unary_adding_operator] term {binary_adding_operator term} 4.4: term ::= factor {multiplying_operator factor} 4.4: factor ::= primary [** primary] | **abs** primary | **not** primary 4.4: * primary ::= numeric_literal | character_literal | string_literal | name | type_conversion | qualified_expression | (expression)	

4.5: logical_operator ::=**and** | **or** | **xor** 4.5: relational_operator ::= = | /= | < | <= | > | >= 4.5: binary_adding_operator ::= + |– |& 4.5: unary_adding_operator ::= + |– 4.5: multiplying_operator ::= * | / | **mod** | **rem** 4.5: highest_precedence_operator ::= ** | **abs** | **not**	4.5: * 4.5: relational_operator ::= = | /= | < | <= | > | >= 4.5: binary_adding_operator ::= + |– |& 4.5: unary_adding_operator ::= + |– 4.5: multiplying_operator ::= * | / | **mod** | **rem** 4.5: *	
4.6: type_conversion ::= subtype_mark(expression) | subtype_mark(name)	4.6: * type_conversion ::= subtype_mark (expression)	
4.7: qualified_expression ::= subtype_mark'(expression) | subtype_mark'aggregate	4.7: qualified_expression ::= subtype_mark'(expression) | subtype_mark'aggregate	
4.8: allocator ::= **new** subtype_indication | **new** qualified_expression	4.8: *	
5.1: sequence_of_statements ::= statement {statement} 5.1: statement ::= {label} simple_statement | {label} compound_statement	5.1: sequence_of_statements ::= statement {statement} 5.1: * statement ::= {label} simple_statement | {label} compound_statement	No (*) before the SPARK rule "statement" and "label " because the two rules are identical in the two syntax.

5.1: simple_statement ::= null_statement | assignment_statement | exit_statement | goto_statement | procedure_call_statement | simple_return_statement | entry_call_statement | requeue_statement | delay_statement | abort_statement | raise_statement | code_statement 5.1: compound_statement ::= if_statement | case_statement | loop_statement l | block_statement | extended_return_statement | accept_statement | select_statement 5.1: null_statement ::= **null**; 5.1: label ::= <<*label*_statement_identifier>> 5.1: statement_identifier ::= direct_name	5.1: * simple_statement ::= null_statement | assignment_statement | procedure_call_statement | exit_statement | return_statement | entry_call_statement | delay_statement 5.1: * compound_statement ::= if_statement | case_statement | loop_statement 5.1: null_statement ::= **null**; 5.1: * label ::= <<*label*_statement_identifier>> 5.1: statement_identifier ::= direct_name	
5.2: assignment_statement ::= *variable*_name := expression;	5.2: + unconstrained_array_assignment ::= *variable*_name := (others => expression); 5.2: * assignment_statement ::= variable_name := expression; + | unconstrained_array_assignment;	

5.3: if_statement ::= **if** condition **then** sequence_of_statements {**elsif** condition **then** sequence_of_statements} [**else** sequence_of_statements] **end if**; 5.3: condition ::= *boolean*_expression	5.3: if_statement ::= **if** condition **then** sequence_of_statements {**elsif** condition **then** sequence_of_statements} [**else** sequence_of_statements] **end if**; 5.3: condition ::= *boolean*_expression	
5.4: case_statement ::= **case** expression **is** case_statement_alternative {case_statement_alternative} **end case**; 5.4: case_statement_alternative ::= **when** discrete_choice_list => sequence_of_statements	5.4: * case_statement ::= **case** expression **is** case_statement_alternative {case_statement_alternative} [**when others** => sequence_of_statements] **end case**; 5.4: case_statement_alternative ::= **when** discrete_choice_list => sequence_of_statements	
5.5: loop_statement ::= [*loop*_statement_identifier:] [iteration_scheme] **loop** sequence_of_statements **end loop** [*loop*_identifier]; 5.5: iteration_scheme ::= **while** condition | **for** loop_parameter_specification 5.5: loop_parameter_specification ::= defining_identifier **in** [**reverse**] discrete_subtype_definition	5.5: loop_statement ::= [*loop*_statement_identifier:] [iteration_scheme] **loop** sequence_of_statements **end loop** [*loop*_identifier]; 5.5: iteration_scheme ::= **while** condition | **for** loop_parameter_specification 5.5: * loop_parameter_specification ::= defining_identifier **in** [**reverse**] *discrete*_subtype_mark [**range** range]	Anonymous case in the For statement. Ex : for I in Integer range 1 .. 10 loop
5.6: block_statement ::= [*block*_statement_identifier:] [**declare** declarative_part] **begin** handled_sequence_of_statements **end** [*block*_identifier];	5.6: *	

5.7: exit_statement ::= **exit** [*loop*_name] [**when** condition];	5.7: * exit_statement ::= **exit** [*simple*_name][**when** condition] ;	
5.8: goto_statement ::= **goto** *label*_name;	5.8: *	
6.1: subprogram_declaration ::= [overriding_indicator] subprogram_specification; 6.1: subprogram_specification ::= procedure_specification | function_specification 6.1: procedure_specification ::= **procedure** defining_program_unit_name parameter_profile 6.1: function_specification ::= **function** defining_designator parameter_and_result_profile 6.1: designator ::= [parent_unit_name.] identifier | operator_symbol 6.1: defining_designator ::= defining_program_unit_name | defining_operator_symbol 6.1: defining_program_unit_name ::= [parent_unit_name .]defining_identifier 6.1: operator_symbol ::= string_literal	6.1: * subprogram_declaration ::= procedure_specification ; procedure_annotation | function_specification ; function_annotation | [overriding_indicator] procedure_specification ; procedure_annotation | [overriding_indicator] function_specification ; function_annotation 6.1: * 6.1: + procedure_specification ::= **procedure** defining_identifier parameter_profile 6.1: + function_specification ::= **function** defining_designator parameter_and_result_profile 6.1: * designator ::= identifier 6.1: * defining_designator ::= defining_identifier 6.1: defining_program_unit_name ::= [parent_unit_name.]defining_identifier 6.1: operator_symbol ::= string_literal	(+) before procedure specificationmust be (*); and before function specification there are nothing because the two syntax are identical.

6.1: defining_operator_symbol ::= operator_symbol 6.1: parameter_profile ::= [formal_part] 6.1: parameter_and_result_profile ::= [formal_part] **return** [null_exclusion] subtype_mark | [formal_part] **return** access_definition 6.1: formal_part ::= (parameter_specification {; parameter_specification}) 6.1: parameter_specification ::= defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression] | defining_identifier_list : access_definition [:= default_expression] 6.1: mode ::= [**in**] | **in out** | **out**	6.1: defining_operator_symbol ::= operator_symbol 6.1: parameter_profile ::= [formal_part] 6.1: parameter_and_result_profile ::= [formal_part] **return** subtype_mark 6.1: formal_part ::= (parameter_specification {; parameter_specification}) 6.1: * parameter_specification ::= defining_identifier_list : mode subtype_mark 6.1: mode ::= [**in**] | **in out** | **out**	
	6.1.1: + procedure_annotation ::= [global_definition] [dependency_relation] [declare_annotation] 6.1.1: + function_annotation ::= [global_definition]	

	6.1.2: + global_definition ::= –# **global** global_modeglobal_variable_list ; { global_modeglobal_variable_list ; } 6.1.2: + global_mode ::= **in** | **in out** | **out** 6.1.2: + global_variable_list ::= global_variable { , global_variable } 6.1.2: + global_variable ::= entire_variable 6.1.2: + entire_variable ::= [*package*_name .] direct_name	
	6.1.2: + dependency_relation ::= –# **derives** [dependency_clause { **&** dependency_clause } [**&** null_dependency_clause]] ; | –# **derives** null_dependency_clause ; 6.1.2: + dependency_clause ::= exported_variable_list **from** [imported_variable_list] 6.1.2: + exported_variable_list ::= exported_variable { , exported_variable } 6.1.2: + exported_variable ::= entire_variable 6.1.2: + imported_variable_list ::= * | [* ,] imported_variable { , imported_variable } 6.1.2: + imported_variable ::= entire_variable 6.1.2: + null_dependency_clause ::= **null from** imported_variable { , imported_variable }	

6.3: subprogram_body ::= [overriding_indicator] subprogram_specification **is** declarative_part **begin** handled_sequence_of_statements **end** [designator];	6.3: * subprogram_body ::= procedure_specification [procedure_annotation] **is** subprogram_implementation | function_specification [function_annotation] **is** subprogram_implementation | [overriding_indicator] procedure_specification [procedure_annotation] **is** subprogram_implementation | [overriding_indicator] function_specification [function_annotation] **is** subprogram_implementation	
	6.3: + subprogram_implementation ::= declarative_part **begin** sequence_of_statements **end designator** ; | **begin** code_insertion **end designator** ; 6.3: + code_insertion ::= code_statement { code_statement }	

6.4: procedure_call_statement ::= *procedure*_name; | *procedure*_prefix actual_parameter_part; 6.4: function_call ::= *function*_name | *function*_prefix actual_parameter_part 6.4: actual_parameter_part ::= (parameter_association {, parameter_association}) 6.4: parameter_association ::= [*formal_parameter*_selector_name =>] explicit_actual_parameter 6.4: explicit_actual_parameter ::= expression | *variable*_name	6.4: * procedure_call_statement ::= *procedure*_name [actual_parameter_part] ; 6.4: * *function*_call ::= *function*_name [actual_parameter_part] 6.4: * actual_parameter_part ::= (parameter_association_list) 6.4: * 6.4: + parameter_association_list ::= named_parameter_association_list |positional_parameter_association_list 6.4: + named_parameter_association_list ::= *formal_parameter_selector*_name => explicit_actual_parameter {, *formal_parameter*_selector_name => explicit_actual_parameter } 6.4: + positional_parameter_association_list ::= explicit_actual_parameter { , explicit_actual_parameter } explicit_actual_parameter ::= expression | *variable*_name	
6.5: simple_return_statement ::= **return** [expression]; 6.5: extended_return_statement ::= **return** defining_identifier : [**aliased**] return_subtype_indication [:= expression] [**do** handled_sequence_of_statements **end return**]; 6.5: return_subtype_indication ::= subtype_indication | access_definition	6.5: * return_statement ::= **return** [expression];	

6.7: null_procedure_declaration ::= [overriding_indicator] procedure_specification **is null**;		
7.1: package_declaration ::= package_specification; 7.1: package_specification ::= **package** defining_program_unit_name **is** {basic_declarative_item} [**private** {basic_declarative_item}] **end** [[parent_unit_name.]identifier]	7.1: * package_declaration ::= [inherit_clause] package_specification ; 7.1: + private_package_declaration ::= [inherit_clause] **private** package_specification ; 7.1: * package_specification ::= **package** defining_program_unit_name package_annotation **is** visible_part [**private** private_part] **end** [parent_unit_name .] identifier 7.1: + visible_part ::= {renaming_declaration } {package_declarative_item } 7.1: + private_part ::= {renaming_declaration } {package_declarative_item } 7.1: + package_declarative_item ::= basic_declarative_item | subprogram_declaration | external_subprogram_declaration	
	7.1.1: + inherit_clause ::= –# **inherit** package_name { , package_name } ;	
	7.1.2: + package_annotation ::= [own_variable_clause [initialization_specification]]	

	7.1.3: + own_variable_clause ::= –# **own** own_variable_specification {own_variable_specification} 7.1.3: + own_variable_specification ::= own_variable_list [: type_mark] [(property_list)]; 7.1.3: + own_variable_list ::= own_variable_modifier own_variable {, own_variable_modifier own_variable }; 7.1.3: +own_variable_modifier ::= mode | **task** | **protected** | **protected in** | **protected out** 7.1.3: +own_variable ::= direct_name 7.1.3: +property_list ::= property {, property} 7.1.3: + property ::= name_property | name_value_property 7.1.3: + name_property ::= **delay** | identifier 7.1.3: + name_value_property ::= identifier => aggregate | expression	
	7.1.4: + initialization_specification ::= –# **initializes** own_variable_list ;	

7.2: package_body ::= **package body** defining_program_unit_name **is** declarative_part [**begin** handled_sequence_of_statements] **end** [[parent_unit_name.]identifier];	7.2: * package_body ::= **package body** defining_program_unit_name [refinement_definition] **is** package_implementation **end** [parent_unit_name.] identifier ; 7.2: + package_implementation ::= declarative_part [**begin** package_initialization] 7.2: + package_initialization ::= sequence_of_statements	
	7.2.1: + refinement_definition ::= –# **own** refinement_clause { & refinement_clause } ; 7.2.1: + refinement_clause ::= subject **is** constituent_list 7.2.1: + subject ::= direct_name 7.2.1: + constituent_list ::= mode constituent {, mode constituent} 7.2.1: + constituent ::= [package_name.] direct_name	
7.3: private_type_declaration ::= **type** defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**; 7.3: private_extension_declaration ::= **type** defining_identifier [discriminant_part] **is** [**abstract**] [**limited** | **synchronized**] **new** *ancestor*_subtype_indication [and interface_list] **with private**;	7.3: * private_type_declaration ::= **type** defining_identifier **is** [**tagged**] [**limited**] **private**; 7.3: * private_extension_declaration ::= **type** defining_identifier [discriminant_part] **is new** *ancestor*_subtype_indication [and interface_list] **with private**;	
8.3.1: overriding_indicator ::= [**not**] **overriding**	8.3.1: overriding_indicator ::= [**not**] **overriding**	

8.4: use_clause ::= use_package_clause | use_type_clause 8.4: use_package_clause ::= **use** package_name {, *package*_name}; 8.4: use_type_clause ::= **use type** subtype_mark {, subtype_mark};	8.4: * 8.4: use_type_clause ::= **use type** subtype_mark {, subtype_mark};	
8.5: renaming_declaration ::= object_renaming_declaration | exception_renaming_declaration | package_renaming_declaration | subprogram_renaming_declaration | generic_renaming_declaration	8.5: * renaming_declaration ::= package_renaming_declaration | subprogram_renaming_declaration	
8.5.1: object_renaming_declaration ::= defining_identifier : [null_exclusion] subtype_mark **renames** *object*_name; | defining_identifier : access_definition **renames** *object*_name;	8.5.1: *	
8.5.2: exception_renaming_declaration ::= defining_identifier : **exception** **renames** *exception*_name;	8.5.2: *	
8.5.3: package_renaming_declaration ::= **package** defining_program_unit_name **renames** *package*_name;	8.5.3: * package_renaming_declaration ::= **package** defining_program_unit_name **renames** par- ent_unit_name.package_direct_name ;	
8.5.4: subprogram_renaming_declaration ::= [overriding_indicator] subprogram_specification **renames** *callable_entity*_name;	8.5.4: * subprogram_renaming_declaration ::= **function** defining_operator_symbol formal_part **return** subtype_mark **renames** package_name.operator_symbol; | function_specification **renames** package_name.function_direct_name; | procedure_specification **renames** pack- age_name.procedure_direct_name;	

8.5.5: generic_renaming_declaration ::= **generic package** defining_program_unit_name **renames** *generic_package*_name; | **generic procedure** defining_program_unit_name **renames** *generic_procedure*_name; | **generic function** defining_program_unit_name **renames** *generic_function*_name;	8.5.5: *	
9.1: task_type_declaration ::= **task type** defining_identifier [known_discriminant_part] [**is** [**new** interface_list **with**] task_definition]; 9.1: single_task_declaration ::= **task** defining_identifier [**is** [**new** interface_list **with**] task_definition]; 9.1: task_definition ::= {task_item} [**private** {task_item}] **end** [*task*_identifier] 9.1: task_item ::= entry_declaration | aspect_clause 9.1: task_body ::= **task body** defining_identifier **is** declarative_part **begin** handled_sequence_of_statements **end** [*task*_identifier];	9.1: task_type_declaration ::= **task type** defining_identifier [known_discriminant_part] task_type_annotation task_definition 9.1: + task_type_annotation ::= moded_global_definition [dependency_relation] [declare_annotation] 9.1: task_definition ::= **is** priority_pragma {apragma} **end** defining_identifier ; 9.1: priority_pragma ::= **pragma Priority** (expression); | **pragma Interrupt_Priority** (expression); 9.1: known_discriminant_part ::= (discriminant_specification {; discriminant_specification}) 9.1: discriminant_specification ::= identifier_list : type_mark 9.1: declare_annotation ::= –# **declare** property_list ;	

9.4: protected_type_declaration ::= **protected type** defining_identifier [known_discriminant_part] **is** [**new** interface_list **with**] protected_definition;	9.4: protected_type_declaration ::= **protected type** defining_identifier [known_discriminant_part] **is** protected_definition;	
9.4: single_protected_declaration ::= **protected** defining_identifier **is** [**new** interface_list **with**] protected_definition;	9.4: *	
9.4: protected_definition ::= { protected_operation_declaration } [**private** { protected_element_declaration }] **end** [*protected*_identifier]	9.4: * protected_definition ::= protected_operation_declaration [**private** protected_element_declaration] **end** defining_identifier; | protected_operation_declaration **private** hidden_part;	
9.4: protected_operation_declaration ::= subprogram_declaration | entry_declaration | aspect_clause	9.4: * protected_operation_declaration ::= priority_pragma entry_or_subprogram {protected_operation} 9.4: * entry_or_subprogram ::= subprogram_declaration | entry_declaration 9.4: * protected_operation ::= apragma | entry_or_subprogram	
9.4: protected_element_declaration ::= protected_operation_declaration | component_declaration	9.4: protected_element_declaration ::= variable_declaration {; variable_declaration}	
9.4: protected_body ::= **protected body** defining_identifier **is** { protected_operation_item } **end** [*protected*_identifier];	9.4: protected_body ::= **protected body** defining_identifier **is** protected_operation_item {protected_operation_item } **end** defining_identifier;	
9.4: protected_operation_item ::= subprogram_declaration | subprogram_body | entry_body | aspect_clause	9.4: protected_operation_item ::= | subprogram_body | entry_body 9.4: protected_body_stub ::= **protected body** defining_identifier **is separate**;	

9.5.2: entry_declaration ::= [overriding_indicator] **entry** defining_identifier [(discrete_subtype_definition)] parameter_profile; 9.5.2: accept_statement ::= **accept** *entry*_direct_name [(entry_index)] parameter_profile [**do** handled_sequence_of_statements **end** [*entry*_identifier]]; 9.5.2: entry_index ::= expression 9.5.2: entry_body ::= **entry** defining_identifier entry_body_formal_part entry_barrier **is** declarative_part **begin** handled_sequence_of_statements **end** [*entry*_identifier]; 9.5.2: entry_body_formal_part ::= [(entry_index_specification)] parameter_profile 9.5.2: entry_barrier ::= **when** condition 9.5.2: entry_index_specification ::= **for** defining_identifier **in** discrete_subtype_definition	9.5.2: * entry_declaration ::= entry_specification ; procedure_annotation 9.5.2: + entry_specification ::= **entry** defining_identifier [formal_part] 9.5.2: * entry_body ::= entry_specification **when** component_identifier procedure_annotation **is** subprogram_implementation	
9.5.3: entry_call_statement ::= *entry*_name [actual_parameter_part];	9.5.3: entry_call_statement ::= *entry*_name [actual_parameter_part];	
9.5.4: requeue_statement ::= **requeue** *entry*_name [**with abort**];		

9.6: delay_statement ::= delay_until_statement | delay_relative_statement 9.6: delay_until_statement ::= **delay until** *delay*_expression; 9.6: delay_relative_statement ::= **delay** *delay*_expression;	9.6: * delay_statement ::= **delay until** Time_expression ;	
9.7: select_statement ::= selective_accept | timed_entry_call | conditional_entry_call | asynchronous_select		
9.7.1: selective_accept ::= **select** [guard] select_alternative { **or** [guard] select_alternative } [**else** sequence_of_statements] **end select**; 9.7.1: guard ::= **when** condition => 9.7.1: select_alternative ::= accept_alternative | delay_alternative | terminate_alternative 9.7.1: accept_alternative ::= accept_statement [sequence_of_statements] 9.7.1: delay_alternative ::= delay_statement [sequence_of_statements] 9.7.1: terminate_alternative ::= **terminate**;		

9.7.2: timed_entry_call ::= **select** entry_call_alternative **or** delay_alternative **end select**; 9.7.2: entry_call_alternative ::= procedure_or_entry_call [sequence_of_statements] 9.7.2: procedure_or_entry_call ::= procedure_call_statement | entry_call_statement		
9.7.3: conditional_entry_call ::= **select** entry_call_alternative **else** sequence_of_statements **end select**;	9.7.3: *	
9.7.4: asynchronous_select ::= **select** triggering_alternative **then abort** abortable_part **end select**; 9.7.4: triggering_alternative ::= triggering_statement [sequence_of_statements] 9.7.4: triggering_statement ::= procedure_or_entry_call | delay_statement 9.7.4: abortable_part ::= sequence_of_statements		
9.8: abort_statement ::= **abort** *task*_name {, *task*_name};		

10.1.1: compilation ::= {compilation_unit} 10.1.1: compilation_unit ::= context_clause library_item | context_clause subunit 10.1.1: library_item ::= [**private**] library_unit_declaration | library_unit_body | [**private**] library_unit_renaming_declaration 10.1.1: library_unit_declaration ::= subprogram_declaration | package_declaration | generic_declaration | generic_instantiation 10.1.1: library_unit_renaming_declaration ::= package_renaming_declaration | generic_renaming_declaration | subprogram_renaming_declaration 10.1.1: library_unit_body ::= subprogram_body | package_body 10.1.1: parent_unit_name ::= name	10.1.1: compilation ::= {compilation_unit} 10.1.1: compilation_unit ::= context_clause library_item | context_clause subunit 10.1.1: * library_item ::= library_unit_declaration | library_unit_body 10.1.1: * library_unit_declaration ::= package_declaration | private_package_declaration | main_program 10.1.1: * 10.1.1: * library_unit_body ::= package_body 10.1.1: parent_unit_name ::= name 10.1.1: + main_program ::= [inherit_clause] main_program_annotation [global_definition [dependency_relation]] subprogram_body 10.1.1: + main_program_annotation ::= –# **main_program**;	

10.1.2: context_clause ::= {context_item} 10.1.2: context_item ::= with_clause | use_clause 10.1.2: with_clause ::= limited_with_clause | nonlimited_with_clause 10.1.2: limited_with_clause ::= **limited** [**private**] **with** *library*_*unit*_name {, *library*_*unit*_name}; 10.1.2: nonlimited_with_clause ::= [**private**] **with** *library*_*unit*_name {, *library*_*unit*_name};	10.1.2: context_clause ::= {context_item} 10.1.2: * context_item ::= with_clause | use_type_clause 10.1.2: * with_clause ::= **with** *library*_package_name {, *library*_package_name}	
10.1.3: body_stub ::= subprogram_body_stub | package_body_stub | task_body_stub | protected_body_stub 10.1.3: subprogram_body_stub ::= [overriding_indicator] subprogram_specification **is separate**;	10.1.3: * body_stub ::= subprogram_body_stub | package_body_stub 10.1.3 * subprogram_body_stub ::= procedure_specification [procedure_annotation] **is separate**; | function_specification [function_annotation] **is separate**; | [overriding_indicator] procedure_specification [procedure_annotation] **is separate** | [overriding_indicator] function_specification [function_annotation] **is separate**; 10.1.3: * package_body_stub ::= **package body** defining_identifier **is separate**;	

10.1.3: package_body_stub ::= **package body** defining_identifier **is separate**; 10.1.3: task_body_stub ::= **task body** defining_identifier **is separate**; 10.1.3: protected_body_stub ::= **protected body** defining_identifier **is separate**; 10.1.3: subunit ::= **separate** (parent_unit_name) proper_body	10.1.3: subunit ::= **separate** (parent_unit_name) proper_body	
11.1: exception_declaration ::= defining_identifier_list : **exception**;	11.1: *	
11.2: handled_sequence_of_statements ::= sequence_of_statements [**exception** exception_handler {exception_handler}] 11.2: exception_handler ::= **when** [choice_parameter_specification:] exception_choice {| exception_choice} => sequence_of_statements 11.2: choice_parameter_specification ::= defining_identifier 11.2: exception_choice ::= *exception*_name | **others**	11.2: *	
11.3: raise_statement ::= **raise**; | **raise** *exception*_name [**with** *string*_expression];	11.3: *	

12.1: generic_declaration ::= generic_subprogram_declaration | generic_package_declaration 12.1: generic_subprogram_declaration ::= generic_formal_part subprogram_specification; 12.1: generic_package_declaration ::= generic_formal_part package_specification; 12.1: generic_formal_part ::= **generic** {generic_formal_parameter_declaration | use_clause} 12.1: generic_formal_parameter_declaration ::= formal_object_declaration | formal_type_declaration | formal_subprogram_declaration | formal_package_declaration	12.1: *	
12.3: generic_instantiation ::= **package** defining_program_unit_name **is new** *generic_package*_name [generic_actual_part]; | [overriding_indicator] **procedure** defining_program_unit_name **is new** *generic_procedure*_name [generic_actual_part]; | [overriding_indicator] **function** defining_designator **is new** *generic_function*_name [generic_actual_part]; 12.3: generic_actual_part ::= (generic_association {, generic_association}) 12.3: generic_association ::= [*generic_formal_parameter*_selector_name =>] explicit_generic_actual_parameter	12.3: * generic_instantiation ::= **function** defining_designator **is new** *generic_function*_name [generic_actual_part] 12.3: generic_actual_part ::= (generic_association {, generic_association}) 12.3: generic_association ::= [*generic_formal_parameter_* *selector*_name =>] explicit_generic_actual_parameter	

12.3: explicit_generic_actual_parameter ::= expression | *variable*_name | *subprogram*_name | *entry*_name | subtype_mark | *package_instance*_name	12.3: explicit_generic_actual_parameter ::= subtype_mark	
12.4: formal_object_declaration ::= defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]; defining_identifier_list : mode access_definition [:= default_expression];	12.4: *	
12.5: formal_type_declaration ::= **type** defining_identifier[discriminant_part] **is** formal_type_definition; 12.5: formal_type_definition ::= formal_private_type_definition | formal_derived_type_definition | formal_discrete_type_definition | formal_signed_integer_type_definition | formal_modular_type_definition |formal_floating_point_definition | formal_ordinary_fixed_point_definition | formal_decimal_fixed_point_definition | formal_array_type_definition | formal_access_type_definition | formal_interface_type_definition	12.5: *	
12.5.1: formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private** 12.5.1: formal_derived_type_definition ::= [**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list] **with private**]	12.5.1: *	

12.5.2: formal_discrete_type_definition ::= (<>) 12.5.2: formal_signed_integer_type_definition ::= **range** <> 12.5.2: formal_modular_type_definition ::= **mod** <> 12.5.2: formal_floating_point_definition ::= **digits** <> 12.5.2: formal_ordinary_fixed_point_definition ::= **delta** <> 12.5.2: formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>	12.5.2: *	
12.5.3: formal_array_type_definition ::= array_type_definition	12.5.3: *	
12.5.4: formal_access_type_definition ::= access_type_definition	12.5.4: *	
12.5.5: formal_interface_type_definition ::= interface_type_definition	12.5.5 *	
12.6: formal_subprogram_declaration ::= formal_concrete_subprogram_declaration | for- mal_abstract_subprogram_declaration 12.6: formal_concrete_subprogram_declaration ::= **with** subprogram_specification [**is** subprogram_default]; 12.6: formal_abstract_subprogram_declaration ::= **with** subprogram_specification **is abstract** [subprogram_default]; 12.6: subprogram_default ::= default_name | <> | **null** 12.6: default_name ::= name	12.6: *	

12.7: formal_package_declaration ::= **with package** defining_identifier **is** **new** *generic_package*_name formal_package_actual_part; 12.7: formal_package_actual_part ::= ([**others** =>] <>) | [generic_actual_part] | (formal_package_association {, formal_package_association} [, **others** => <>]) 12.7: formal_package_association ::= generic_association | *generic_formal_parameter*_selector_name => <>	12.7: *	
13.1: aspect_clause ::= attribute_definition_clause | enumeration_representation_clause | record_representation_clause | at_clause 13.1: local_name ::= direct_name | direct_name'attribute_designator | *library_unit*_name	13.1: * representation_clause ::= attribute_definition_clause | enumeration_representation_clause | record_representation_clause | at_clause 13.1: local_name ::= direct_name | direct_name'attribute_designator | *library_unit*_name	In SPARK, just renaming aspect clause to representation clause.

13.3:
attribute_definition_clause ::=
for local_name'attribute_designator
use expression;
| **for** local_name'attribute_designator
use name;
	13.3:
	* attribute_definition_clause ::=
	for local_name'attribute_designator
	use simple_expression;	

13.4: enumeration_representation_clause ::= **for** *first_subtype*_local_name **use** enumeration_aggregate; 13.4: enumeration_aggregate ::= array_aggregate	13.4: enumeration_representation_clause ::= **for** *first_subtype*_local_name **use** enumeration_aggregate; 13.4: enumeration_aggregate ::= array_aggregate	
	13.5: at_clause ::= **for** *simple*_name **use at** *simple*_expression ;	For Ada, see RM J.7

13.5.1: record_representation_clause ::= **for** *first_subtype*_local_name **use** **record** [mod_clause] {component_clause} **end record**; 13.5.1: component_clause ::= *component*_local_name **at** position **range** first_bit .. last_bit;	13.5.1: * record_representation_clause ::= **for** *first_subtype*_local_name **use** **record** [mod_clause] {component_clause} **end record**; 13.5.1: component_clause ::= *component*_local_name **at** position **range** first_bit .. last_bit;	No difference, so no * record representation clause
13.5.1: position ::= *static*_expression 13.5.1: first_bit ::= *static*_simple_expression 13.5.1: last_bit ::= *static*_simple_expression	13.5.1: * position ::= *static*_simple_expression 13.5.1: first_bit ::= *static*_simple_expression 13.5.1: last_bit ::= *static*_simple_expression 13.5.1: mod_clause ::= **at mod** *simple*_expression	
13.8: code_statement ::= qualified_expression;	13.8: code_statement ::= qualified_expression;	
13.12: restriction ::= *restriction*_identifier | *restriction_parameter*_identifier => restriction_parameter_argument 13.12: restriction_parameter_argument ::= name | expression		
J.3: delta_constraint ::= **delta** *static*_expression [range_constraint] J.7: at_clause ::= **for** direct_name **use at** expression; J.8: mod_clause ::= **at mod** *static*_expression;		