ASDL Reference Manual
Version 3.0

John Reppy
jhr@cs.chicago.edu

Revised: September 2018

Contents

Chapter 1

Introduction

The *Abstract Syntax Description Lanuguage* (**ASDL**) is a language designed to describe the tree-like data structures in compilers. Its original purpose to provide a method for compiler components written in different languages to interoperate. **ASDL** makes it easier for applications written in a variety of programming languages to communicate complex recursive data structures.

`asdlgen` is a tool that takes **ASDL** descriptions and produces implementations of those descriptions in a variety of languages. **ASDL** and `asdlgen` together provide the following advantages

- Concise descriptions of important data structures.
- Automatic generation of data structure implementations for `asdlgen`-supported languages.
- Automatic generation of functions to read and write the data structures to disk in a machine and language independent way.

ASDL descriptions describe the tree-like data structures such as abstract syntax trees (ASTs) and compiler intermediate representations (IRs). Tools such as `asdlgen` automatically produce the equivalent data structure definitions for the supported languages. `asdlgen` also produces functions for each language that read and write the data structures to and from a platform and language independent sequence of bytes. The sequence of bytes is called a *pickle*.

ASDL was originally developed in the 1990's by Daniel Wang as part of the *National Compiler Infrastructure* project at Princeton University. Tha **ASDL** implementation has not kept up with the significant changes in many of its target languages (*e.g.*, C++, HASKELL, JAVA, *etc.*), so it was time for a rewrite.[1] Version 3.0 of **ASDL** and `asdlgen` is a complete reimplementation of the system, with the primary purpose of supporting the SML/NJ compiler. As such, it currently only supports generating picklers in SML and modern C++, although other languages may be added as time permits.

1.1 Changes From Version 2.0

The following is a list of the major changes from the 2.0 version of **ASDL** and `asdlgen`:

- The primitive types were extended and changed. The `bool` type was added, the type name of arbitrary precision integers was changed to `integer`, and the types `int` and `uint` were added to represent small integers.

[1] Version 2.0 of **ASDL** is still available from `https://sourceforge.net/projects/asdl` and the 2.0 version of the manual (converted to LaTeX is included in the documentation of this system.

- Various changed were made to the binary encoding of pickles.
- Currently only two target languages are supported: SML and C++.
- The generated C++ code targets the 2011 standard and uses the C++ STL (*e.g.*, `std::vector<>` for **ASDL** sequences).
- Data can be pickled/unpickled to/from memory, as well as files. This change affects the requirements for implementing primitive modules.
- Alias-type definitions were added to the ASDL syntax.
- Include directives were added to support splitting specifications into multiple files (and sharing of common specifications).

Chapter 2

ASDL Syntax

This section describes the syntax of the input language to `asdlgen`. The syntax is described using EBNF notation. Literal terminals are typeset in bold and enclosed in single quotes. Optional terms are enclosed in square brackets and terms that are repeated zero or more times are enclosed in braces. Each section describes a fragment of the syntax and its meaning.

2.1 Lexical Tokens

The lexical conventions for **ASDL** are given in Figure 2.1. ASDL is a case-sensitive language and, furthermore, classifies identifiers into lower-case (⟨*lc-id*⟩) and upper-case (⟨*uc-id*⟩) identifiers. Type identifiers are lower-case, while constructor identifiers are upper-case. Module and field identifiers can be either upper or lower-case.

Comments begin with "`--`" and continue to the end of the line.

Verbatim text is denoted by ⟨*text*⟩ and can be specified in one of two ways. Either by an initial "`:`" followed by a sequence of ⟨*text-character*⟩s that continues to the end of the line or by a "`%%`" terminated by a "`%%`" at the beginning of a line by itself. Text include using the "`:`" notation will have trailing and leading whitespace removed.

ASDL has the following keywords:

'**alias**' '**attributes**' '**import**', '**include**' '**module**' '**primitive**' '**view**'

Note that it is allowed to use a keyword as an identifier wherever a ⟨*lc-id*⟩ is permitted.

2.2 File Syntax

An **ASDL** file consists of one or more ⟨*definition*⟩s possibly preceded by '**include**' directives. A definition specifies either a module (see Section 2.3), primitive module (see Section 2.5), or view (see Section 2.6).

Include directives allow one to split a large **ASDL** specification into multiple files, while allowing `asdlgen` to check references from one module to another. `asdlgen` will parse included files, but will not generate code for the definitions in included files. Also, included files will only be parsed once.

⟨*upper*⟩ ::= '**A**' | ... | '**Z**'

⟨*lower*⟩ ::= '**a**' | ... | '**z**'

⟨*alpha*⟩ ::= '_' | ⟨*upper*⟩ | ⟨*lower*⟩

⟨*alpha-num*⟩ ::= ⟨*alpha*⟩ | '**0**' | ... | '**9**'

⟨*lc-id*⟩ ::= ⟨*lower*⟩ { ⟨*alpha-num*⟩ }

⟨*uc-id*⟩ ::= ⟨*upper*⟩ { ⟨*alpha-num*⟩ }

⟨*id*⟩ ::= ⟨*lc-id*⟩ | ⟨*uc-id*⟩

⟨*typ-id*⟩ ::= ⟨*lc-id*⟩

⟨*con-id*⟩ ::= ⟨*uc-id*⟩

⟨*comment*⟩ ::= '**--**' ⟨*text-character*⟩ ⟨*end-of-line*⟩

⟨*text*⟩ ::= '**:**' { ⟨*text-character*⟩ } ⟨*end-of-line*⟩
| '**%%**' { ⟨*text-character*⟩ | ⟨*end-of-line*⟩ } ⟨*end-of-line*⟩ '**%%**'

Figure 2.1: Lexical rules for **ASDL** terminals

⟨*file*⟩ ::= { '**include**' ⟨*text*⟩ } ⟨*definition*⟩ { ⟨*definition*⟩ }

⟨*definition*⟩ ::= ⟨*module*⟩
| ⟨*primitive-module*⟩
| ⟨*view*⟩

Figure 2.2: **ASDL** file syntax

2.3 Module Syntax

Figure 2.3 gives the syntax for modules. An **ASDL** module declaration consists of the keyword '**module**' followed by an identifier, an optional set of imported modules, and a sequence of type definitions enclosed in braces.

For example the following example declares modules `A`, `B`, and `C`. `B` imports types from `A`. `C` imports types from both `A` and `B`. Imports cannot be recursive; for example, it is an error for `B` to import `C`, since `C` imports `B`.

```
module A { ... }
module B (import A) { ... }
module C (import A
          import B) { ... }
```

To refer to a type imported from another module the type must *always* be qualified by the module name from which it is imported. The following declares two different types called "`t`." One in module `A` and one in module `B`. The type "`t`" in module `B` defines a type "`t`" that recursively mentions itself and also references the type "`t`" imported from module `A`.

⟨*module*⟩ ::= '**module**' ⟨*id*⟩ [⟨*imports*⟩] '**{**' ⟨*definitions*⟩ '**}**'

⟨*imports*⟩ ::= '**(**' { '**import**' ⟨*id*⟩ ['**alias**' ⟨*id*⟩] } '**)**'

Figure 2.3: **ASDL** module syntax

⟨*type-definition*⟩ ::= ⟨*typ-id*⟩ '**=**' ⟨*type*⟩

⟨*type*⟩ ::= ⟨*alias-type*⟩ | ⟨*sum-type*⟩ | ⟨*product-type*⟩

⟨*alias-type*⟩ ::= ⟨*typ-exp*⟩

⟨*product-type*⟩ ::= ⟨*fields*⟩

⟨*sum-type*⟩ ::= ⟨*constructor*⟩ { '**|**' ⟨*constructor*⟩ } ['**attributes**' ⟨*fields*⟩]

⟨*constructor*⟩ ::= ⟨*con-id*⟩ [⟨*fields*⟩]

⟨*fields*⟩ ::= '**(**' { ⟨*field*⟩ '**,**' } ⟨*field*⟩ '**)**'

⟨*field*⟩ ::= ⟨*typ-exp*⟩ [⟨*id*⟩]

⟨*typ-exp*⟩ ::= [⟨*id*⟩ '**.**'] ⟨*typ-id*⟩ ['**?**' | '*****' | '**!**']

Figure 2.4: **ASDL** type definition syntax

```
module A { t = ... }
module B (import A) { t = T(A.t, t) | N  ... }
```

2.4 Type Definitions

The syntax of type definitions is given in Figure 2.4. A type defintion begins with a type identifier, which is the name of the type. The name must be unique within the module, but the order of definitions is unimportant. When translating type definitions from a module they are placed in what would be considered a module, package, or name-space of the same name. If the output language does not support such features and only has one global name space the module name is used to prefix all the globally exported identifiers.

Type definitions are either alias types, which bind a name to a type expression; product types, which are simple record definitions; or sum type, which represent a discriminated union of possible values. Unlike sum types, product types cannot form recursive type definitions, but they can contain recursively declared sum types.

2.4.1 Alias Types

Alias types are the simplest form of type definition. They provide a way to give a name to a type or type expression, similar to SML's **type** and C++'s **typedef** constructs.

2.4.2 Type expressions

A type expression (⟨*typ-exp*⟩) consists of a possibly qualified type name followed by an optional type operator. If the specified type is an ASDL primitive type or is defined in the current module, then its name is not qualified; all other types defined outside the current module must be qualified by their module name (or module alias).

The type operators are:

- option (‘`?`’), which specifies either zero or one value of the specified type.
- sequence (‘`*`’), which specifies a sequence of zero or more values of the specified type, or
- shared (‘`!`’), which specifies that a value is shared across multiple points in the data structure (*i.e.*, the structure has a DAG shape instead of just a tree).

Note that while at most one type operator is allowed in a type expression, one can use alias types to combine two or more operators. For example, a sequence of optional integers could be defined by:

```
int_opt = integer?
int_opt_seq = int_opt*
```

2.4.3 ASDL Primitive Types

There are seven pre-defined primitive types in ASDL, which are available without qualification:

`bool` describes Boolean values.

`int` describes signed-integer values that are representable in 30 bits (*i.e.*, in the range -2^{29} to $2^{29}-1$).

`uint` describes unsigned-integer values that representable in 30 bits (*i.e.*, in the range 0 to $2^{30}-1$).

`integer` describes arbitrary-precision signed-integer values.

`natural` describes arbitrary-precision unsigned-integer values.

`string` describes length encoded strings of 8-bit characters.

`identifier` describes strings with fast equality testing analogous to Lisp symbols.

2.4.4 Product Types

Product types are record or tuple declarations. They consist of a non-empty sequence of fields separated by commas enclosed in parenthesis. For example

```
pair_of_ints = (int, int)
```

declares a new type `pair_of_ints` that consists of two integers.

2.4.5 Sum Types

Sum types are the most useful types in ASDL. They provide concise notation used to describe a type that is the tagged union of a finite set of other types. Sum types consists of a series of constructors separated by a vertical bar. Each constructor consist of a constructor identifier followed by an optional sequence of fields similar to a product type.

Constructor names must be unique within the module in which they are declared. Constructors can be viewed as functions who take some number of arguments of arbitrary type and create a value belonging to the sum type in which they are declared. For example

```
module M {
  sexpr = Int(int)
        | String(string)
        | Symbol(identifier)
        | Cons(sexpr, sexpr)
        | Nil
}
```

declares that values of type `sexpr` can either be constructed from an `int` using the `Int` constructor or a `string` from a `String` constructor, an `identifier` using the `Symbol` constructor, or from two other `sexpr` using the `Cons` constructor, or from no arguments using the `Nil` constructor. Notice that the `Cons` constructor recursively refers to the `sexpr` type. **ASDL** allows sum types to be mutually recursive. Recursion, however, is limited to sum types defined within the same module.

Sum Types as Enumerations

Sum types that consist completely of nullary constructors are often treated specially and translated into static constants of a enumerated value in languages that support them. For example, the following **ASDL** specification:

```
module Op {
  op = PLUS | MINUS | TIMES | DIVIDE
}
```

Is translated into the following C++ code:

```
namespace M {
    enum class op {
        PLUS = 1, MINUS, TIMES, DIVIDE
    };
}
```

2.4.6 Fields

A field consists of a type expression followed by an optional label. The fields of a product or sum type must either all be labeled or unlabeled. We use *record* to refer to products of labeled fields and *tuple* to products of unlabeled fields. Labels aid in the readability of descriptions and are used by `asdlgen` to name the fields of records and classes for languages. For example the declarations

```
module Tree {
  tree = Node(int, tree, tree)
       | EmptyTree
}
```

can also be written as

```
module Tree {
  tree = Node(int value, tree left, tree right)
       | EmptyTree
}
```

When translating the first definition without labels into C++ one would normally get

```
namespace Tree {
    ...
    class Node : public tree {
      public:
        Node (int p_value, tree * p_left, tree * p_right)
            : tree(tree::_con_Node), _v_value(p_value), _v_left(p_left),
            _v_right(p_right)
        { }
        ~Node ();
        void encode (asdl::outstream & os);
        int get_value () { return this->_v_value; }
        void set_value (int v) { this->_v_value = v; }
        tree * get_left () { return this->_v_left; }
        void set_left (tree * v) { this->_v_left = v; }
        tree * get_right () { return this->_v_right; }
        void set_right (tree * v) { this->_v_right = v; }
      private:
        int _v_value;
        tree * _v_left;
        tree * _v_right;
    };
    ...
} // namespace Tree
```

with labels one would get

```
namespace Tree {
    $\cdots$
    struct Node : public tree {
        int value;
        tree *left;
        tree *right;
        Node (int v1, tree *v2, tree * v3)
          : tree(tree::_Node), value(v1), left(v2), right(v3) { }
        ~Node () { }
    };
    $\cdots$
}
```

For the SML target, product types without labels are translated to tuples, while those with labels are translated to records.

2.4.7 Attribute Fields

A sum-type definition may optionally be followed by a list of attribute fields, which provide a concise way to specify fields that are common to all of the constructors of a sum type. For example, the definition

⟨*primitive-module*⟩ ::= '**primitive**' '**module**' ⟨*id*⟩ '**{**' { ⟨*id*⟩ } '**}**'

Figure 2.5: **ASDL** primitive module syntax

```
module M {
  pos = (string file, int linenum, int charpos)
  sexpr = Int(int)
        | String(string)
        | Symbol(identifier)
        | Cons(sexpr, sexpr)
        | Nil
        attribute(pos)
}
```

adds a field of type `pos` to all the constructors in `sexpr`. One can think of an `attribute` annotation as syntactic sugar for just including the extra fields at the *beginning* of each constructor's fields. For example, the above definition can be viewed as syntactic sugar for

```
module M {
  pos = (string file, int linenum, int charpos)
  sexpr = Int(pos, int)
        | String(pos, string)
        | Symbol(pos, identifier)
        | Cons(pos, sexpr, sexpr)
        | Nil(pos)
}
```

Note that this interpretation implies that attribute fields are labeled if, and only if, all of the constructor fields are labeled.

Attribute fields are treated specially when translating to some targets. For example in C++ code, the attribute field is defined in the base class for the sum type.

2.5 Primitive Modules

Primitive modules (see Figure 2.5) provide a way to introduce abstract types that are defined outside of **ASDL** and which have their own pickling and unpickling code. For example, we might want to include GUIDs (16-byte globally-unique IDs) in our pickles. We can do so by first defining a primitive module `Prim`:

```
primitive module Prim { guid }
```

Then, depending on the target language, we define supporting code to read and write guids from the byte stream. In SML, we would define three modules:

1. **structure** `Prim` that defines the representation of the `guid` type.
2. **structure** `PrimPickle` that defines functions for pickling/unpickling a GUID to/from memory.
3. **structure** `PrimPickleIO` that defines functions for reading and writing GUIDs on binary I/O streams.

The SML implementation of these modules could be written as follows:

```
structure Prim : sig
    type guid
  end = struct
    type guid = GUID.guid
  end

structure PrimPickle : sig
    val guidEncode : Word8Buffer.buf * Prim.guid -> unit
    val guidDecode : Word8VectorSlice.slice -> Prim.guid * Word8VectorSlice.slice
  end = struct
    val guidSize = 16
    fun guidEncode (buf, guid) = Word8Buffer.addVec(buf, GUID.toBytes guid)
    fun guidDecode slice = let
          val
    fun guidToBytes = GUID.toBytes
    fun guidFromBytes { input } = (case input 16
           of NONE => raise Fail "bogus GUID"
            | (SOME v) => (case GUID.fromBytes v
             of NONE => raise Fail "bogus GUID"
              | SOME g => g
            (* end case *))
          (* end case *))
  end

structure PrimPickleIO : sig
    val guidInput : BinIO.instream -> guid
    val guidOutput : BinIO.outstream * guid -> unit
  end = struct
    fun guidInput strm = (case GUID.fromBytes(BinIO.inputN(strm, 16))
           of NONE => raise Fail "bogus GUID"
            | (SOME g) => g
          (* end case *))
    fun guidOutput (strm, g) = BinIO.output(strm, GUID.toBytes g)
  end
```

(assuming that the `GUID` module implements the application's representation of GUIDs).

For C++, a primitive module requires a corresponding header file that declares the primitive types and instances of the overloaded `<<` and `>>` operators on the primitive types. These declarations should all be in a **namespace** with the type name of the primitive module. For example, the module from above would require the provision of a `Prim.hxx` header file that contained something like the following code:

```
#include <iostream>
#include "guid.hxx"

namespace Prim {

    typedef GUID::guid guid;

    std::istream &operator>> (std::istream &is, guid &g);
    std::ostream &operator<< (std::ostream &os, guid const &g);

}
```

(assuming that the `guid.hxx` header defines the application's representation of GUIDs).

⟨*view*⟩ ::= '**view**' ⟨*id*⟩ '**{**' { ⟨*view-entry*⟩ } '**}**'

⟨*view-entry*⟩ ::= ⟨*view-entities*⟩ '**<=**' ⟨*view-properties*⟩
| '**<=**' ⟨*id*⟩ '**{**' { ⟨*view-entity*⟩ ⟨*text*⟩ } '**}**'

⟨*view-entities*⟩ ::= ⟨*view-entity*⟩
| '**{**' { ⟨*view-entity*⟩ } '**}**'

⟨*view-entity*⟩ ::= '**<file>**'
| '**module**' ⟨*id*⟩
| ⟨*id*⟩ '**.**' ⟨*typ-id*⟩ ['**.**' '*****']
| ⟨*id*⟩ '**.**' ⟨*typ-id*⟩ '**.**' ⟨*con-id*⟩

⟨*view-properties*⟩ ::= ⟨*id*⟩ ⟨*text*⟩
| '**{**' { ⟨*id*⟩ ⟨*text*⟩ } '**}**'

Figure 2.6: **ASDL** view syntax

2.6 View Syntax

A view defines how an **ASDL** specification is translated to a target. Each of the supported targets (*e.g.*, SML or C++) has a default view, but it is possible to customize the translation using ⟨*view*⟩ definitions. The syntax of view declarations is given in Figure 2.6. This section covers the syntax of views, but leaves the semantics to Chapter 3.

2.6.1 Basic View Syntax

Views are named and consist of series of entries. In its basic form, a view entry consists of a ⟨*view-entity*⟩, which specifies a file, module, type, or constructor entity, and a view property, which is a name-value pair that is associated with the entity.

⟨*view-entry*⟩ ::= ⟨*view-entity*⟩ '**<=**' ⟨*id*⟩ ⟨*text*⟩

The meaning of an entry is to associate the specified view property with the specified view entity.

There can be multiple views with the same name. The entries of two views with the same name are merged and consist of the union of the entries in both. It is an error, for two views of the same name to assign different values to the same property of an entity.

2.6.2 View Entry Derived Forms

To make it easier to specify view entries, **ASDL** generalizes the basic syntax to remove some of the redundancy of the basic syntax. First, it is possible to specify multiple view entities on the left-hand-side of the '**<=**' symbol. Likewise, it is possible to specify multiple view properties on the right-hand-side of the '**<=**' symbol.

It is also possible to assign different values to different entities for a fixed property using the syntax.

⟨*view-entry*⟩ ::= '**<=**' ⟨*id*⟩ '**{**' { ⟨*view-entity*⟩ ⟨*text*⟩ } '**}**'

Here the property name is given first, followed by a sequence of view-entity-value pairs.

Lastly, **ASDL** allows a '**.***' suffix to be added to sum-type entities. This suffix means that the entity specifies the set of all of the constructors of the type.

Chapter 3

Views

Views provide a general mechanism to customize the output of `asdlgen`. Views allow description writers to annotate modules, types, and constructors with directives or properties that are interpreted by `asdlgen`. Currently `asdlgen` properties that allow for the following mechanisms are supported:

- Inclusion of arbitrary user code in the resulting output.
- Automatic coercion of specific types into more efficient user defined representations.
- Addition of extra user defined attributes and initialization code.
- Control over how the names of types, constructors, and modules names are mapped into the output language to resolve style issues and name space conflicts.
- Control over the tag values for sum types.
- Addition of documentation that describes the meaning of types constructors and modules.

3.1 Interpretation of View Entry Values

See the chapter on Input Syntax for details on view the syntax and some basic view terminology. The view syntax associates an arbitrary string whose interpretation depends on the property it is assigned too. Currently there is a small set of standard interpretations.

integer An integral number in decimal notation.

string A raw ASCII string.

boolean A boolean value (either "`true`" or "`false`").

qualified identifier A possibly qualified identifier (*e.g.*, "`M.t`" or "`t`"). Qualified identifiers are language independent identifiers that are translated to the appropriate output language in a uniform way. For example `M.t` would appear as `M::t` in C++ and as `M.t` in SML.

3.2 Adding User Code

It is useful to be able to add arbitrary user code to the modules produced by `asdlgen`. Modules have six properties that can be set to allow the addition of user-code strings.

`interface_prologue`
: Include text verbatim after the introduction of the base environment, but before any type defined in the module interface.

`interface_epilogue`
: Include text verbatim after all types defined in the module interface have been defined.

`implementation_prologue`
: Include text verbatim after the introduction of the base environment, but before any other implementation code is defined.

`implementation_epilogue`
: Include text verbatim after all definitions defined in the module implementation.

`suppress`
: Default value is false. Do not produce any code for this module, assume it's implementation is written by hand. It's often a good idea to first generate code and then set the flag, so the generated code can be used as stubs for the user implementation.

`is_library`
: Default value is false. If true assume all types can be used as lists or options and generate any needed code, rather then generating list and option code on demand. Useful for generating stubs.

The precise meaning of interface and implementation for the different languages is as follows

C++ The interface is the `.hxx` file and the implementation is the `.cxx` file.

SML The interface is the generated signature the implementation is the structure.

3.3 Other Properties

`doc_string`
: All entities have this property. Its value is interpreted as a string. Currently only the *`--doc`* command recognizes the property. It includes the property value in the HTML documentation produced for the module.

`source_name`
: All entities have this property. The value is interpreted as a string. Choose a different name for the type constructor or module in the output code. The name has no case restrictions. This is particularly useful when producing Java code on Windows NT/95 since the file system is case insensitive and types and constructors that differ only in case will cause problems.

`user_attribute`
: Property of types only. The value is interpreted as a qualified identifier. Add a field called `client_data` as an attribute to the type. The value is the qualified identifier that represents

an arbitrary user type of the field. The `client_data` field is ignored by the pickling code and does not appear in constructors. This property is currently only recognized when outputting C.

`user_init`
Property of types only. The value is interpreted as a qualified identifier. Call the function specified by the value before returning a the data structure created by a constructor function. This property is currently only recognized when outputting C.

`base_class`
Property of types only. The value is interpreted as a qualified identifier. The name of the class from which all classes generated for that type should inherit from. This property is recognized only when outputting C++ and Java.

`reader`
Property of types only. The value is interpreted as a qualified identifier. Replace the body of the read pickle function for this type with a call to a function with the proper arguments.

`writer`
Property of types only. The value is interpreted as a qualified identifier. Replace the body of the writer pickle function for this type with a call to a function with the proper arguments.

`enum_value`
Property of constructors only. The value is interpreted as an integer. Use this integer value as the `internal` tag value for the constructor. The external pickle tag remains unchanged. This property is recognized only when outputting C, C++, and Java.

3.4 Choosing a Different Representation

```
module IntMap {
  int_map = (int size,entries map)
  entries = (entry* entries)
  entry   = (int key, int value)
}
```

The above is one possible abstract description of a mapping from integers to integers. It would be more efficient to *implement* such a mapping as a binary tree. Described as with the ASDL definition below.

```
module IntMap {
  int_map = (size int, map tree)
     tree = Node(int key, int value, tree left, tree right)
          | Empty
}
```

Although this is a much more efficient representation it exposes implementation details. If we decided to change the implementation of `int_maps` to use a hash table the all other clients that use our type will have to be updated.

The view properties `natural_type`, `natural_type_con`, `wrapper`, and `unwrapper` provide a general mechanism to choose a different more efficient representation through coercion functions. All of these properties apply to types only and are interpreted as qualified identifiers.

natural_type
: The type to use in place of the original type in all the resulting code. Supported by all output languages.

natural_type_con
: A unary type constructor to apply to the old type to get a new type to use in all the resulting code; *e.g.*, ref in ML to make a type mutable. Supported by ML and Haskell. *Support for C++ templates will be added in the near future.*

wrapper
: A function to convert the new type to the old type when writing the pickle. Supported by all output languages.

unwrapper
: A function to convert the old type to the new type when reading the pickle. Supported by all output languages.

When using `natural_type` and `natural_type_con` the automatically generated type definitions for the original type still remain, but all other references to the original type in constructors, picklers, and other type definitions that referred to it are replaced with the new type. The original definition must remain to support pickling of the type. Pickling is achieved by appropriately coercing the new type to the old type and vice versa with functions specified by `wrapper` and `unwrapper` properties.

3.4.1 Examples

```
module Slp {

 real = (int mantissa, int exp)
...
  exp = Id(identifier id)
      | Num(int v)
      | Op(exp lexp,binop?,exp rexp)
       attributes (real? value)
...

}

view C {

-- represent reals as a double

  Slp.exp <= {
   natural_type : my_real
   wrapper      : real2my_real
   unwrapper    : my_real2real
  }

  module Slp <= {
  interface_epilogue : typedef double my_real_ty

  implementation_prologue
%%
  my_real_ty real2my_real(Slp_real_ty x) {
    /* hairy code to actually do this */
  }
```

```
  Slp_real_ty my_real2real(my_real_ty x) {
    /* hairy code to actually do this */
  }
%%
  }
}

view SML {
   -- unpickle exp trees as a mutable type
    Slp.exp <= {
       natural_type_con : ref
       wrapper          : !
       unwrapper        : ref
    }

}
```

Chapter 4

Code Interface

In this section, we describe the default translation of ASDL definitions to target languages and describe some of the runtime assumptions that users need to be aware of when using the generated code.

4.1 Translation to SML

The translation from an **ASDL** specification to SML code is straightforward. **ASDL** modules map to SML structures, **ASDL** product types map to either tuples or records, and **ASDL** sum types map the SML datatypes. Table 4.1 summarizes this translation. If an **ASDL** identifier conflicts with an SML keyword or pervasive identifier, then the translation adds a trailing prime character (`'`) to the identifier.

For an **ASDL** module *M*, we generate two SML signatures and three SML structures:

```
structure M = struct ... end
signature M_PICKLE = sig ... end
structure MPickle : M_PICKLE = struct ... end
signature M_PICKLE_IO = sig ... end
structure MPickleIO : M_PICKLE_IO = struct ... end
```

where *M* structure contains the type definitions for the **ASDL** specification, `MPickle` structure implements functions to convert between the types and byte vectors, and the `MPickleIO` structure implements functions to read and write pickles from binary files.

For an ASDL source file `f.asdl`, `asdlgen` will produce five SML source files:

`f.sml`	contains type definition structures (*e.g.*, `structure` *M*)
`f-pickle.sig`	contains memory-pickler signatures (*e.g.*, `signature` *M*`_PICKLE`)
`f-pickle.sml`	contains memory-pickler structures (*e.g.*, `structure` *M*`Pickle`)
`f-pickle-io.sig`	contains file-pickler signatures (*e.g.*, `signature` *M*`_PICKLE_IO`)
`f-pickle-io.sml`	contains file-pickler structures (*e.g.*, `structure` *M*`PickleIO`)

4.1.1 CM support

The SML/NJ Compilation Manager (CM) knows about ASDL files (as of version 110.84). If one specifies "`foo.asdl`" in the file list of a `.cm` file, CM will infer the generation of the five SML files as described above.

Table 4.1: Translation of **ASDL** types to SML

ASDL type	**SML type**
Named types (T)	($\hat{T}$)
`bool`	`bool`
`int`	`int`
`uint`	`word`
`integer`	`IntInf.int`
`string`	`string`
`identifier`	`Atom.atom`
t	t
M`.`t	M`.`t
Type expressions (τ)	($\hat{\tau}$)
T	$\hat{T}$
T`?`	$\hat{T}$ `option`
T`*`	$\hat{T}$ `list`
Product types (ρ)	($\hat{\rho}$)
`(`τ_1`, ..., `τ_n`)`	$\hat{\tau}_1$ `*` $\cdots$ `*` $\hat{\tau}_n$
`(`τ_1 f_1`, ..., `τ_n f_n`)`	`{`f_1 `:` $\hat{\tau}_1$`, ...,` f_n `:` $\hat{\tau}_n$`}`
Type definitions	
t `=` ρ	**`type`** t `=` $\hat{\rho}$
t `=` C_1 `|` $\cdots$ `|` C_n	**`datatype`** t `=` C_1 `|` $\cdots$ `|` C_n
t `=` C_1`(`ρ_1`)` `|` $\cdots$ `|` C_n`(`ρ_n`)`	**`datatype`** t `=` C_1 **`of`** $\hat{\rho}_1$ `|` $\cdots$ `|` C_n **`of`** $\hat{\rho}_n$

4.2 Translation to C++

The translation of an **ASDL** specification to C++ is more complicated than for SML. For each **ASDL** module, we define a corresponding C++ namespace.

4.2.1 Memory management

4.3 The Rosetta Stone for Sum Types

In languages that have algebraic data types sum types are equivalent to the `datatype` and `data` declarations in ML and Haskell. In Algol like languages they are equivalent to tagged `unions` in C or variant records in Pascal. In class based object oriented languages sum types are equivalent to an abstract base class that represents the type of a family of subclasses one for each constructor of the type. The previous example written in ML would be

```
structure M =
  struct
    datatype sexpr
      = Int of (int)
      | String of (string)
      | Symbol of (identifier)
      | Cons of (sexpr * sexpr)
      | Nil
  end
```

and in C++ it translates to

```
namespace M {

    struct sexpr {
        enum tag {
            _Int, _String, _Symbol, _Cons, _Nil
        };
        tag _tag;
        sexpr (tag t) : _tag(t) { }
        virtual ~sexpr ();
    };

    struct Int : public sexpr {
        int _v1;
        Int (int v) : sexpr(sexpr::_Int), _v1(v) { }
        ~Int () { }
    };

    struct String : public sexpr {
        std::string _v1;
        String (const char *v) : sexpr(sexpr::_String), _v1(v) { }
        String (std::string const &v) : sexpr(sexpr::_String), _v1(v) { }
        ~String () { }
    };

    struct Symbol : public sexpr { ... };

    struct Cons : public sexpr { ... };

    struct Nil : public sexpr { ... };
```

Table 4.2: Translation of **ASDL** types to C++

ASDL type	**C++ type**
Named types (T)	($\hat{T}$)
`bool`	`bool`
`int`	`int`
`uint`	`unsigned int`
`integer`	`asdl::integer`
`string`	`std::string`
`identifier`	`asdl::identifier`
t	$\begin{cases} t & \text{if } t \text{ is an } \texttt{enum} \text{ type} \\ t\texttt{*} & \textit{otherwise} \end{cases}$
M`.`t	M`::`t
Type expressions (τ)	($\hat{\tau}$)
T	$\hat{T}$
T`?`	`asdl::option<` $\hat{T}$ `>`
T`*`	`std::vector<` $\hat{T}$ `>`
Product types (ρ)	($\hat{\rho}$)
`(`τ_1`, ..., `τ_n`)`	$\hat{\tau}_1$ `_v1; ...` $\hat{\tau}_n$ `_vn`
`(`τ_1 f_1`, ...,` τ_n f_n`)`	$\hat{\tau}_1$ $\_f_1$`; ...` $\hat{\tau}_n$ $\_f_n$
Type definitions	
t `=` ρ	`struct` t `{` $\hat{\rho}$ `};`
t `=` C_1 `|` $\cdots$ `|` C_n	`class enum` t `{` C_1`, ...,` C_n `};`

t `=` C_1`(`ρ_1`)` `|` $\cdots$ `|` C_n`(`ρ_n`)`	`class` t `{` $\cdots$ `};`
	`class` C_1 `: public` t `{`
	`  private:` $\hat{\rho}_1$
	`  ...`
	`};`
	`...`
	`class` C_n `: public` t `{`
	`  private:` $\hat{\rho}_n$
	`  ...`
	`};`

```
}
```

Chapter 5

Pickles

One of the most important features of `asdlgen` is that it automatically produces functions that can read and write the data structures it generates to and from a platform and language independent external representation. This process of converting data structures in memory into a sequence of bytes on the disk is referred to as *pickling*. Since it is possible to generate data structures and pickling code for any of the supported languages from a single **ASDL** specification, `asdlgen` provides an easy and efficient way to share complex data structures among these languages.

The **ASDL** pickle format requires that both the reader and writer of the pickler agree on the type of the pickle. Other than constructor tags for sum types, there is no explicit type information in the pickle. In the case of an error the behavior is undefined. It is also important that the pickling/unpickling to/from files, that the files be opened in binary mode to prevent line feed translations from corrupting the pickle.

5.1 Binary Pickle Format

Since **ASDL** data structures have a tree-like form, they can be represented linearly with a simple prefix encoding.

5.1.1 Primitive types

bool

Boolean values are represented by `0` (false) or `1` (true) and are encoded in one byte.

int

The `int` type provides 30-bits of signed precision encoded in one to four bytes. The top two bits of the first byte (bit 6–7) specify the number of additional bytes in the encoding and bit 5 specifies the sign of the number. Thus values in the range -32 to 31 can be encoded in one byte, -8192 to 8191 in two bytes, *etc.* A negative number n is represented as the positive number $-(n+1)$.

uint

The `uint` type provides 30-bits of unsigned precision encoded in one to four bytes. As with the `int` type, the top two bits of the first byte specify the number of additional bytes in the encoding. Thus values in the range 0 to 63 can be encoded in one byte, 0 to 16383 unsigned in two bytes, *etc.*

integer
: The **ASDL** `integer` type is represented with a variable-length, big-endian, signed-magnitude encoding. The high bit of each byte indicates if the byte is the last byte of the encoding. The bit 6 of the most significant byte is used to determine the sign of the value. Thus, numbers in the range of -63 to 63 are encoded in one byte. Numbers outside of this range require an extra byte for every seven bits of precision required.

string
: Strings are represented with a length-header that describe how many more 8-bit bytes follow for the string and then the data for the string in bytes. The length-header is encoded as a `uint` value, thus strings are limited to 1,073,741,823 characters.

identifier
: Identifiers are represented as if they were strings.

5.1.2 Product types

The fields of a product type are encoded sequentially (left to right) without any initial tag.

sequence types
: Sequence types are represented with an integer length-header followed by that many values of that type. The length-header is encoded as a `uint` value, thus sequences are limited to at most 1,073,741,823 items.

option types
: The encoding of optional values depends on the base type. For sum types with more than one constructor, the special tag value of zero is used to denote an empty value and non-zero values are interpreted as the constructor's tag. For any other base type, there is an initial byte that is either one or zero. A zero indicates that the value is empty and no more data follows. A one indicates that the next value is the value of the optional value.

5.1.3 Enumeration types

5.1.4 Sum types

Sum types begin with a unique tag to identify the constructor followed by the fields of the constructor. The tag is encoded as either one (`tag8`) or two (`tag16`) bytes, depending on the number of constructors in the type. Tag values are assigned in order of constructor definition starting from one (the value zero is used to encode empty option values). Fields are packed left to right based of the order in the definition. If there are any attribute values associated with the type, they are packed left to right after the tag but before other constructor fields.

There are two special cases for sum-type encodings. If the sum types is an enumeration with only one constructor, then no space is used to encode the value, and if the type has a single constructor with fields, then it is encoded without a tag (*i.e.*, like a product type).

5.1.5 Alias types

5.1.6 primitive types

User-defined primitive types are pickled/unpickled by user-provided functions (see Section 2.5).

5.2 S-expression Format

It is also possible to generate a text-based representation of pickles in S-Expression syntax.

bool

numbers
The ASDL numeric types (`int`, `uint`, and `integer`) are represented by decimal literals.

string and identifier
These values are represented by string literals.

enumeration types

sum types

sequence types

option types

primitive types
User-defined primitive types are not yet supported in S-Expression form.

5.3 XML Pickle Format

Chapter 6

Usage

Synopsis

```
asdlgen command [ options ] files ...
```

Where *command* is one of

`help`	Print information about the `asdlgen` tool to the standard output.
`version`	Print the version of of asdlgen to the standard output.
`c++` or `cxx`	Generate C++
`sml`	Generate Standard ML
`check`	Check correctness of inputs, but do not generate output

Description

`asdlgen` reads the set of *files*, which contain ASDL module and view declarations.

Common Options

Options common to all the commands include

`-n`
: Do not write any output files. Instead write the list of files that would have been written to standard out.

`--output-directory=`*dir* or `-d` *dir*
: Specify the output directory to place the generated files. By default the output will be placed in the same directory as the input file from which it was produced.

`--pickler=`*name* or `-p` *name*
: Specifies which kind of pickler to generate. See Chapter 5 for details.

Command-specific Options

All the commands that produce source code as output offer a different command option to select the default base environment. The base environment is the set of the initial definitions available

to the code. It defines the set of primitive types and functions used by the generated code. For example using the option `--base-include=my-base.h` when generating C code will insert the directive

```
#include "my-base.h"
```

in the appropriate place so the resulting code will use the definitions found in `my-base.h` rather than the default set of primitive types. Unless there is a need to globally redefine the primitive types changing the base environment should be avoided. The actual option names vary depending on the output language.

See Chapter 4 for a more detailed description about the interfaces to the default set of primitive types and functions.

Options for C++

`--base-include=`*file*

Specify the name of the C++header file that defines the primitive **ASDL** types and functions. The default value is `asdl/asdl.hxx`.

Chapter 7

Document history

ASDL and `asdlgen` (originally named `asdlGen`) were developed as part of the National Compiler Infrastructure Project at Princeton University in the late 1990's [1]. The original version of this manual was written by Dan Wang as part of that project. As part of reimplementing and modernizing **ASDL**, John Reppy converted this manual to LaTeX and did some light editing. The original implementation of `asdlGen` and of this manual can be found at `http://asdl.sourceforge.net`.

Here is a history of changes to **ASDL**, `asdlgen`, and this manual. The changes are indexed by SML/NJ release numbers.

SML/NJ 110.84

First release of ASDL 3.0; see Section 1.1 for differences from previous version.

Bibliography

[1] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, Berkeley, CA, USA, October 1997. USENIX Association.