ASDL 3.0 Reference Manual

John Reppy
`jhr@cs.chicago.edu`

Revised: July 2020

ii

Contents

Chapter 1

Introduction

The *Abstract Syntax Description Lanuguage* (ASDL) is a language designed to describe the tree-like data structures used in compilers. Its original purpose was to provide a method for compiler components written in different languages to interoperate [1], but it has also been used to support communicating information between separate runs of a compiler. ASDL makes it fairly easy 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. That 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]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.

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.

- Various changes 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 the 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 initial-lower-case (⟨*lc-id*⟩) and initial-upper-case (⟨*uc-id*⟩) identifiers. Type identifiers are initial-lower-case, while constructor identifiers are initial-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 included 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.

$$\langle upper \rangle \ ::= \ \text{`A'} \mid ... \mid \text{`Z'}$$

$$\langle lower \rangle \ ::= \ \text{`a'} \mid ... \mid \text{`z'}$$

$$\langle alpha \rangle \ ::= \ \text{`\_'} \mid \langle upper \rangle \mid \langle lower \rangle$$

$$\langle alpha\text{-}num \rangle \ ::= \ \langle alpha \rangle \mid \text{`0'} \mid ... \mid \text{`9'}$$

$$\langle lc\text{-}id \rangle \ ::= \ \langle lower \rangle \ \{ \ \langle alpha\text{-}num \rangle \ \}$$

$$\langle uc\text{-}id \rangle \ ::= \ \langle upper \rangle \ \{ \ \langle alpha\text{-}num \rangle \ \}$$

$$\langle id \rangle \ ::= \ \langle lc\text{-}id \rangle \mid \langle uc\text{-}id \rangle$$

$$\langle typ\text{-}id \rangle \ ::= \ \langle lc\text{-}id \rangle$$

$$\langle con\text{-}id \rangle \ ::= \ \langle uc\text{-}id \rangle$$

$$\langle comment \rangle \ ::= \ \text{`--'} \ \{ \ \langle text\text{-}character \rangle \ \} \ \langle end\text{-}of\text{-}line \rangle$$

$$\langle text \rangle \ ::= \ \text{`:'} \ \{ \ \langle text\text{-}character \rangle \ \} \ \langle end\text{-}of\text{-}line \rangle$$
$$\mid \ \text{`\%\%'} \ \{ \ \langle text\text{-}character \rangle \mid \langle end\text{-}of\text{-}line \rangle \ \} \ \langle end\text{-}of\text{-}line \rangle \ \text{`\%\%'}$$

Figure 2.1: Lexical rules for ASDL terminals

$$\langle file \rangle \ ::= \ \{ \ \text{`include'} \ \langle text \rangle \ \} \ \langle definition \rangle \ \{ \ \langle definition \rangle \ \}$$

$$\langle definition \rangle \ ::= \ \langle module \rangle$$
$$\mid \ \langle primitive\text{-}module \rangle$$
$$\mid \ \langle view \rangle$$

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*⟩] '**{**' { ⟨*type-definition*⟩ } '**}**'

⟨*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 ($\langle$*typ-exp*$\rangle$) 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 defined by a non-empty sequence of fields separated by commas enclosed in parenthesis. 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 declaration

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

defines the tuple type `pair_of_ints` that consists of two integers, whereas the declaration

```
size = (int width, int height)
```

defines the record type `size` that consists of two labeled fields: `width` and `height`. Note that ASDL requires that if any field in a product type has a label, then all of them must have labels.

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

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 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, 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
    };
}
```

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

Figure 2.5: ASDL primitive module syntax

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

```
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 two modules:

1. **structure** Prim that defines the representation of the guid type.

2. **structure** PrimPickle that defines functions for pickling/unpickling a GUID using an imperative stream API.

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 read_guid : (unit -> Word8.word) -> unit -> Prim.guid
    val write_guid : (Word8.word -> unit) -> Prim.guid -> unit

  end = struct

    val guidSize = 16
    fun read_guid getByte () =
        GUID.fromBytes(Word8Vector.tabulate(guidSize, fn _ => getByte()))
    fun write_guid putByte guid = let
        Word8Vector.app putByte (GUID.toBytes guid)

  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).

$$\langle\mathit{view}\rangle \ ::= \ \text{`\texttt{view}'} \ \langle\mathit{id}\rangle \ \text{`\{'} \ \{ \ \langle\mathit{view\text{-}entry}\rangle \ \} \ \text{`\}'}$$

$$\langle\mathit{view\text{-}entry}\rangle \ ::= \ \langle\mathit{view\text{-}entities}\rangle \ \text{`\texttt{<=}'} \ \langle\mathit{view\text{-}properties}\rangle$$
$$| \ \ \text{`\texttt{<=}'} \ \langle\mathit{id}\rangle \ \text{`\{'} \ \{ \ \langle\mathit{view\text{-}entry}\rangle \ \langle\mathit{text}\rangle \ \} \ \text{`\}'}$$

$$\langle\mathit{view\text{-}entities}\rangle \ ::= \ \langle\mathit{view\text{-}entity}\rangle$$
$$| \ \ \text{`\{'} \ \{ \ \langle\mathit{view\text{-}entity}\rangle \ \} \ \text{`\}'}$$

$$\langle\mathit{view\text{-}entity}\rangle \ ::= \ \text{`\texttt{<file>}'}$$
$$| \ \ \text{`\texttt{module}'} \ \langle\mathit{id}\rangle$$
$$| \ \ \langle\mathit{id}\rangle \ \text{`.'} \ \langle\mathit{typ\text{-}id}\rangle \ [ \ \text{`.'} \ \text{`\texttt{*}'} \ ]$$
$$| \ \ \langle\mathit{id}\rangle \ \text{`.'} \ \langle\mathit{typ\text{-}id}\rangle \ \text{`.'} \ \langle\mathit{con\text{-}id}\rangle$$

$$\langle\mathit{view\text{-}properties}\rangle \ ::= \ \langle\mathit{id}\rangle \ \langle\mathit{text}\rangle$$
$$| \ \ \text{`\{'} \ \{ \ \langle\mathit{id}\rangle \ \langle\mathit{text}\rangle \ \} \ \text{`\}'}$$

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 $\langle\mathit{view}\rangle$ 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 $\langle\mathit{view\text{-}entity}\rangle$, 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.

$$\langle\mathit{view\text{-}entry}\rangle \ ::= \ \langle\mathit{view\text{-}entity}\rangle \ \text{`\texttt{<=}'} \ \langle\mathit{id}\rangle \ \langle\mathit{text}\rangle$$

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.

$$\langle\mathit{view\text{-}entry}\rangle \ ::= \ \text{`\texttt{<=}'} \ \langle\mathit{id}\rangle \ \text{`\{'} \ \{ \ \langle\mathit{view\text{-}entity}\rangle \ \langle\mathit{text}\rangle \ \} \ \text{`\}'}$$

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 Views and View Entities

Currently, there are two views supported by `asdlgen`: `cxx` for when generating C++ code and `sml` for when generating Standard ML code.

A *view entity* specifies the scope to which a view entry is applied. The entities are:

- The *file* entity, written "`<file>`," which specifies the entire input file.

- A *module* entity, written "`module` `modid`," which specifies the module `modid`.

- A *type* entity, written "`modid.tyid`," which specifies the type `tyid` in the module `modid`.

- A *constructor* entity, written "`modid.tyid.conid`," which specifies the constructor `conid` that belongs to type `tyid` in the module `modid`. It is also possible to specify all of the constructors of a type by using the syntax "`modid.tyid.*`."

13

3.2   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.

names A non-empty list of comma-separated identifiers.

In the discussion below, we specify the interpretation of the view-entry values, the views that accept them, and the kinds of view entities that they may be applied to.

3.2.1   Overriding the Default Names

The view mechanism provides a fair amount of flexibility for overriding the default names used for modules and functions in the generated code.

file_pickler_name *identifier*	`sml` / Module

Specifies the name to give to the target module that implements file pickling. Does not apply to the C⁺⁺ target, since memory and file pickling is combined.

memory_pickler_name *identifier*	`sml` / Module

Specifies the name to give to the target module that implements memory pickling. Does not apply to the C⁺⁺ target, since memory and file pickling is combined.

name *identifier*	All / All

All entities have this property. The value is interpreted as an identifier that overrides the name for the file, module, type, or data constructor in the output code.

pickler_name *identifier*	`c++,sml` / Module

Specifies the name to give to the target module that implements both file and memory pickling. For the SML target, this affects the name of the pickler signature, but not the pickler structures, since different modules are generated for memory and file pickling.

sexp_pickle_name *identifier*	`sml` / Module

Specifies the name to give to the target module that implements S-Expression pickling.

3.2.2 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 to the generated modules.[1]

`interface_prologue` *text*	`c++,sml` / File,Module

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

`interface_epilogue` *text*	`c++,sml` / File,Module

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

`implementation_prologue` *text*	`c++,sml` / File,Module

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

`implementation_epilogue` *text*	`c++,sml` / File,Module

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

`suppress` *names*	`c++,sml` / File,Module

    Specifies which aspects of the generated code should be suppressed (*i.e.*, not generated). The allowed names are:

- `types` — suppresses the generation of the type definitions for the entity.
- `pickler` — suppresses the generation of the pickler code for the entity.
- `unpickler` — suppresses the generation of the unpickler code for the entity.

    It is also possible to specify "`none`," which is the default and means that nothing is suppressed, and "`all`," which means that all code generation for the entity is suppressed. It is often a good idea to first generate code and then set this entry, so that the generated code can be used as stubs for the user implementation.

`private_code` *text*	`c++` / Type,Con

    Add the text to the class generated for the type or constructor with private scope.

`protected_code` *text*	`c++` / Type,Con

    Add the text to the class generated for the type or constructor with protected scope.

`public_code` *text*	`c++` / Type,Con

    Add the text to the class generated for the type or constructor with public scope.

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

C++ The interface is the `.hxx` file and the implementation is the `.cxx` file. Code that is specified for the file entity will appear outside any namespace, whereas code that is specified for a module entity will appear inside the module's namespace.

---

[1]In the case of a target language like SML, where multiple modules are generated, the code is added to the base module that contains the generated type definitions.

SML The interface is the generated signature and the implementation is the structure. Code that is
specified for the file entity will appear at top-most level, whereas code that is specified for a
module entity will appear inside the corresponding signature or structure body.

3.2.3 Choosing a Different Representation

The ASDL module

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

is one possible abstract description of a mapping from integers to integers. Such an implementation
is not particularly efficient; we might prefer to use binary-search trees for more efficient lookups.
We can easily describe such a data structure in ASDL

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

but this description exposes implementation details and prevents the use of existing library code.
Furthermore, changing the representation to use a hash table would require that all clients be
changed.

asdlgen supports four properties – natural_type, natural_type_con, wrapper, and
unwrapper – to allow clients to use a specialized representation for ASDL types.[2]

natural_type *identifier*	c++,sml / Type

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

natural_type_con *identifier*	c++,sml / Type

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 SMLto make a type mutable. *Support for C++ templates will be added in
the near future.*

wrapper *identifier*	c++,sml / Type

Specifies the name of the function to convert the pickle type to the natural type when reading
the pickle. The interface code for this function will be generated, but the implementation must
be provided using the implementation_prologue or implementation_epilogue prop-
erties.

---

[2]Primitive modules are another way to solve this problem, but they require all of the pickling and unpickling code be
provided by the user.

unwrapper *identifier*	`c++,sml` / Type

Specifies the name of the function to convert the natural type to the pickle type when writing the pickle. The interface code for this function will be generated, but the implementation must be provided using the `implementation_prologue` or `implementation_epilogue` properties.

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.

For example, we could use the SML/NJ Library's `IntRedBlackMap` structure to implement the `int_map` type as follows:

```
view sml {
  module IntMap <= {
    interface_prologue : type int_map = int IntRedBlackMap.map
    implementation_prologue
%%
    structure IntMap = IntRedBlackMap
    type int_map = int IntMap.map
%%
    implementation_epilogue
%%
    fun wrap_int_map ({map={entries}, ...} : int_map_pkl) =
          List.foldl
            (fn ({key, value}, imap) => IntMap.insert(imap, key, value))
              IntMap.empty
                entries
    fun unwrap_int_map (imap : int_map) = {
            size = IntMap.numItems imap,
            map = {entries = IntMap.foldri
              (fn (k, v, entries) => {key = k, value = v} :: entries)
                []
                  imap}
          }
%%
  }
  IntMap.int_map <= {
    name : int_map_pkl
    natural_type : int_map
    wrapper : wrap_int_map
    unwrapper : unwrap_int_map
  }
}
```

In this view, we rename `int_map` to `int_map_pkl` and add a type definition for `int_map` to both the interface (signature) and implementation (structure). We also add definitions of the wrapper and unwrapper functions. The generated code is

```
structure IntMap : sig
    type int_map = int IntRedBlackMap.map
    type entry = {key : int, value : int}
    type entries = {entries : entry list}
    type int_map_pkl = {size : int, map : entries}
    type map_pair = int_map * int_map
    val wrap_int_map : int_map_pkl -> int_map
    val unwrap_int_map : int_map -> int_map_pkl
  end = struct
    structure IntMap = IntRedBlackMap
    type int_map = int IntMap.map
    type entry = {key : int, value : int}
    type entries = {entries : entry list}
    type int_map_pkl = {size : int, map : entries}
    type map_pair = int_map * int_map
    fun wrap_int_map ({map={entries}, ...} : int_map_pkl) = ...
    fun unwrap_int_map (imap : int_map) = ...
  end
```

Note that we had to define the `int_map` type in the prologue so that the definition of the `int_map` type was well-defined.

3.2.4   Pickling Existing SML Datatypes

Because ASDL types have a direct correspondence with SML types, it is often the case that an existing SML type in your code is identical to what ASDL would generate for a given specification. In such a case, we can suppress the generation of the type definitions for the module using the "`suppress types`" view entity for the module in question. `asdlgen` will still generate the pickler modules, but not the type definitions.

3.2.5   Other Properties

| **doc_string** *text* | / |

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.

| **user_attribute** *identifier* | / |

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** *identifier* | / |

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

`base_class` *identifier* | / Type |

    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⁺⁺.

`reader` *identifier* | / Type |

    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` *identifier* | / T |

    ype 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` *integer* | c++ / Con |

    Property of constructors only. Use this integer value as the *internal* tag value for the constructor. The external pickle tag remains unchanged.

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 an SML signature and several SML structures:

```
structure M = struct ... end
signature M_PICKLE = sig ... end
structure MMemoryPickle : M_PICKLE = struct ... end
structure MFilePickle : M_PICKLE = struct ... end
structure MSExpPickle : M_PICKLE = struct ... end (* optional *)
```

where M structure contains the type definitions for the ASDL specification, MMemoryPickle structure implements functions to convert between the types and byte vectors, and the MFilePickle structure implements functions to read and write pickles from binary files. The optional MSExpPickle structure implements functions to read and write textual pickles in S-Expression syntax.[1] This module is generated when the "--sexp" option is specified (see see Section 5.2 for more details).

For an ASDL source file f.asdl, asdlgen will produce four 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)

---

[1]Currently, only output of S-Expression pickles is implemented.

21

`f-memory-pickle.sml`
> contains memory-pickler structures (*e.g.*, `structure M`MemoryPickle)

`f-file-pickle.sml`
> contains file-pickler structures (*e.g.*, `structure M`FilePickle)

`f-sexp-pickle.sml`
> contains the optional S-Expression pickler structures (*e.g.*, `structure M`SExpPickle).
> This file is only generated when the "`--sexp`" command-line option is specified.

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.

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   Translation of named types

The following table summarizes how type names are mapped to C++ type expressions.

Named ASDL type (T)	C++ type $\widehat{T}$
bool	**bool**
int	**int**
uint	**unsigned int**
integer	asdl::integer
string	std::string
identifier	asdl::identifier
t	$\begin{cases} t & \text{if } t \text{ is an } \textbf{enum} \text{ type} \\ t\text{*} & otherwise \end{cases}$
$M.t$	M::t

In the subsequent discussion, we write $\widehat{T}$ to denote the C++ type that corresponds the the ASDL named type T.

4.2.2   Translation of type expressions

The translation of ASDL type expressions formed by applying a type operator to a named type T is described in the following table.

Table 4.1: Translation of ASDL types to SML

ASDL type	**SML type**
Named types (T)	$(\widehat{T})$
`bool`	`bool`
`int`	`int`
`uint`	`word`
`integer`	`IntInf.int`
`string`	`string`
`identifier`	`Atom.atom`
t	t
$M.t$	$M.t$
Type expressions (τ)	$(\widehat{\tau})$
T	$\widehat{T}$
T?	$\widehat{T}$ `option`
T*	$\widehat{T}$ `list`
Product types (ρ)	$(\widehat{\rho})$
$(\tau_1,\ \ldots,\ \tau_n)$	$\widehat{\tau}_1\ \star\ \cdots\ \star\ \widehat{\tau}_n$
$(\tau_1\ f_1,\ \ldots,\ \tau_n\ f_n)$	$\{f_1\ :\ \widehat{\tau}_1,\ \ldots,\ f_n\ :\ \widehat{\tau}_n\}$
Type definitions	
t = ρ	`type` t = $\widehat{\rho}$
t = C_1 | $\cdots$ | C_n	`datatype` t = C_1 | $\cdots$ | C_n
t = $C_1(\rho_1)$ | $\cdots$ | $C_n(\rho_n)$	`datatype` t = C_1 `of` $\widehat{\rho}_1$ | $\cdots$ | C_n `of` $\widehat{\rho}_n$

ASDL type expression (τ)	C++ type $\widehat{\tau}$
T	$\widehat{T}$
$T?$	$\begin{cases} \texttt{asdl::option<}\widehat{T}\texttt{>} & \text{if } t \text{ is an } \textbf{enum} \text{ type} \\ \widehat{T} & \textit{otherwise} \end{cases}$
$T\star$	$\texttt{std::vector<}\ \widehat{T}\ \texttt{>}$

Notice that the for option types, we use `nullptr` to represent the empty option for those ASDL types that are represented by C++ pointer types. For other ASDL types, which are not represented by pointers, we wrap the type with the template class "`option<>`" from the ASDL library. This class provides methods for testing if an option is empty (`isEmpty` and for getting the value when it is not (`valOf`).

4.2.3   Translation of type definitions

The translation of type definitions depends on the form of the right-hand-side of the definition. For product types, we map the fields of the product to a sequence of C++ member declarations (as described below) enclosed in a C++ **struct** type. For enumerations, we use C++ scoped enumerations to represent the type. For other sum types, we define an abstract base class for the type with subclasses for each constructor. This translation is summarized in the following table, where we are using ρ to represent the fields of an ASDL product type and $\widehat{\rho}$ for its translation:

ASDL type definition	C++ type
$t = \rho$	`struct` t `{` $\hat{\rho}$ `};`
$t = C_1 \mid \cdots \mid C_n$	`class enum` t `{` C_1`,` `...,` C_n `};`

$t = C_1(\rho_1) \mid \cdots \mid C_n(\rho_n)$	`class` t `{` $\cdots$ `};`
	`class` C_1 `: public` t `{`
	  `private:` $\hat{\rho}_1$
	  `...`
	`};`
	`...`
	`class` C_n `: public` t `{`
	  `private:` $\hat{\rho}_n$
	  `...`
	`};`

The translation of product types is described by the following table:

ASDL product type (ρ)	C++ type $\widehat{\rho}$
$(\tau_1,\ \ldots,\ \tau_n)$	$\hat{\tau}_1$ `_v1;` $\ldots$ $\hat{\tau}_n$ `_vn`
$(\tau_1\ f_1,\ \ldots,\ \tau_n\ f_n)$	$\hat{\tau}_1$ `_`f_1`;` $\ldots$ $\hat{\tau}_n$ `_`f_n

4.2.4   Pickling and unpickling operations

The code generator for the C++ view uses a mix of methods, static methods, and overloaded functions to implement the pickler and unpickler operations.

4.2.5 Memory management

4.2.6 Makefile support

Currently there is no support for generating makefile dependencies or rules, but it may be added in a future release.

4.3 The Rosetta Stone for Sum Types

For languages that support algebraic data types, `asdlgen` maps sum types directly to the language's mechanism (*e.g.*, **datatype** declarations in SML). For class-based object-oriented languages, like C++, `asdlgen` maps sum types to abstract base classes and the constructors to individual subclasses. The previous example written in SML 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 () { }
    };
```

```cpp
struct Symbol : public sexpr { ... };

struct Cons : public sexpr { ... };

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

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*.[1] 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 when pickling/un-pickling 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. This encoding is used by both the memory and file picklers and is described in this section.

5.1.1 Primitive types

bool

> Boolean values are represented by `1` (false) or `2` (true) and are encoded in one byte. We reserve the value zero to represent the empty value for the type `bool?`.

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 $-(n + 1)$.

---

[1]It is also called *serialization*.

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.

tags

> Tags are an internal type that ASDL uses to represent enumerations and tagged sums. If the number of distinct tags required for a type is less than 256, then the tag is represented as a single byte. Otherwise, the tag value is encoded as a `uint` value.

5.1.2   Product types

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

5.1.3   Enumeration types

Enumeration types are represented by a set of tag values, one per constructor on the type. Tag values are assigned in order of constructor definition starting from one. As with the `bool` type, the value zero is used to encode empty option values.

   If the enumeration type has only a single constructor, then it is implicit in the encoding (*i.e.*, no space is used to represent it).

5.1.4   Sum types

Non-enumeration sum types begin with a unique tag to identify the constructor followed by the fields of the constructor. Tag values are assigned in order of constructor definition starting from one (the value zero is used to encode empty option values). As with product types, fields are packed left to right based on 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.

   If the sum type has a single constructor with fields, then it is encoded without a tag (*i.e.*, like a product type).

5.1.5 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.

5.1.6 Option types

The encoding of optional values depends on the base type. For sum types with more than one constructor (including `bool`), 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.

5.1.7 Shared types

Shared types change the representation of a pickle from a tree to a DAG (Directed Acyclic Graph). A shared value is the pickle is either a *definition* occurrence (*i.e.*, the first occurrence of that value), or a *back reference* (*i.e.*, subsequent occurrences of the value). A shared value is represented by an integer followed by the encoding of the value itself. For a definitional occurrence, the integer will be zero, and for subsequent occurrences, the integer will count back to the shared definition (*e.g.*, one maps to the most recent definition, two to the one before it, *etc.*). For unpickling, this representation only requires that the unpickler maintain a stack per shared type. Definitions get pushed onto the stack and back references extract existing values from the stack.

Note that shared types are currently the only feature of ASDL that are *stateful*; *i.e.*, that pickling/unpickling a value depends on previous values in the stream.

5.1.8 Alias types

Alias types use the encoding of their definition.

5.1.9 User-defined 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. Primitive values are represented as literals, enumerations are represented as quoted symbols, and structured values are represented with a parenthesized expression of the form

$$(\; op \;\; v'_1 \;\; \cdots \;\; v'_n \;)$$

where op is an identifier that defines the structure of the value and the v'_1, ..., v'_n are the encodings of the sub-values.

5.2.1   Primitive types

bool
>   Boolean values are mapped to the literals `#t` and `#f`.

numbers
>   Values of the ASDL numeric types (`int`, `uint`, and `integer`) are represented by decimal literals.

string and identifier
>   These values are represented by string literals.

5.2.2   Product types

A product value $(f_1, \ldots, f_n)$ is encoded as the S-expression $(n\text{-tuple } f'_1 \cdots f'_n)$, where the f'_i are the encoded fields of the value. Unlabeled fields are directly represented by their value, whereas a field with label l and value v is encoded as $(l \; v')$, where v' is the encoding of the field's value.

5.2.3   Sum types

Nullary constructors are mapped to quoted symbols, while non-nullary constructors are mapped to an S-expression with the constructor name as the operator and the

5.2.4   Sequence types

A sequence of values $v_1, \ldots, v_n$ is encoded as $(\ast \, v'_1 \cdots v'_n)$, where the v'_i are the encoded fields of the sequence.

5.2.5   Option types

An empty option value is represented as $(?)$, while a non-empty option value with contents v is represented as $(? \, v')$, where v' is the encoding of the contents.

5.2.6   Alias types

Alias types use the encoding of their definition.

5.2.7   User-defined primitive types

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

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.

`--gen=`*names*

> Specifies the components to generate. The value *names* is a list of comma-separated names from the following list:
>
> * "`types`" – generate the type definitions from the ASDL specification.

- "`memory`" – generate the memory pickler
- "`file`" – generate the file pickler
- "`sexp`" – generate the S-Expression pickler (SML only).

The default is "`types,memory,file.`" Alternatively, *names* can be "`none`," which means that no output is generated. Specifying "`none`" is different than using the "`-n`" option, because it does not cause the list of files to be printed.

Note: currently the "`--gen`" option only affects the SML command; it is ignored by the C**++** command.

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.hxx` when generating C**++** code will insert the directive

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

in the appropriate place so the resulting code will use the definitions found in `my-base.hxx` 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`.

Options for Standard ML

`--cm=`*file*
>   Generate a CM file for the pickler; this will define a CM library. Note that if the ASDL specification includes primitive modules, these will be included in the list of exported structures, but the supporting source files will have to be added to the CM file by hand.

`--mlb=`*file*
>   Generate an MLB file for the pickler. Note that if the ASDL specification includes primitive modules, these will be included in the list of exported structures, but the supporting source files will have to be added to the MLB file by hand.

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 the original manual to LaTeX and did some light editing (this version of the old manual can be found in the `doc/manual-2.0` directory). The original implementation of `asdlGen` 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 2021.1

Reorganized the SML library to separate input and output into separate structures. For example, the "`ASDLFilePickle`" structure is now implemented as the pair of structures: "`ASDLFileReadPickle`" for input and "`ASDLFileWritePickle`" for output. The structure "`ASDLFilePickle`" is provided as a combined structure that provides support for both input and output.

SML/NJ 110.99

Fixed various inconsistencies between the implementation (both C++ and SML) and the documented representation.

SML/NJ 110.98

Initial support for the sharing type operator has been added, but it is incomplete.

The semantics of the `suppress` view item has been changed to allow more fine-grained control. Instead of a boolean flag, it now takes a list of aspects to suppress.

General improvements to the documentation.

SML/NJ 110.86

The interface to the file and memory picklers was unified: the "`encode`" was changed to "`write`," "`decode`" was changed to "`read`" in the memory pickler, and the read operations were made imperative. Also added support for writing to an S-Expression textual format and improved the documentation.

Changed the "`--pickler`" option to "`--gen`" and made it take a list of targets.

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.