The Zephyr Abstract Syntax Description Language

Daniel C. Wang Andrew W. Appel Jeff L. Korn
Christopher S. Serra
Department of Computer Science, Princeton University, Princeton, NJ, 08544
{danwang,appel,jlk}@cs.princeton.edu, csserra@cs.wisc.edu

Abstract

The Zephyr[1] Abstract Syntax Description Language (ASDL) describes the abstract syntax of compiler intermediate representations (IRs) and other tree-like data structures. Just as the lexical and syntactic structures of programming languages are described with regular expressions and context free grammars, ASDL provides a concise notation for describing the abstract syntax of programming languages. Tools can convert ASDL descriptions into the appropriate data-structure definitions and functions to convert the data-structures to or from a standard flattened representation. This makes it easier to build compiler components that interoperate.

Although ASDL lacks subtyping and inheritance, it is able to describe the Stanford University Intermediate Format (SUIF) compiler IR, originally implemented in C++. We have built a tool that converts ASDL into C, C++, Java, and ML data-structure definitions and conversion functions. We have also built a graphical browser-editor of ASDL data structures. ASDL shares features found in many network interface description languages (IDLs), algebraic data types, and languages such as ASN.1 and SGML. Compared to other alternatives ASDL is simple and powerful. This document describes ASDL in detail and presents an initial evaluation of ASDL.

1 Introduction

Reusable components make it easy to build compilers to test new compilation techniques. The components of a compiler communicate with each other through an intermediate representation (IR), which is a description of a program suitable for optimization and analysis. If compiler components can exchange compatible IRs they can interoperate.

To interoperate, components need an implementation of an IR and a way to transmit IR values to other components. A simple way to transmit IR values across different components is to read and write the IR to a file in a standard format. These files are called pickles, and conversion to pickles is called pickling or marshaling [BNOW93]. Since different compiler research groups program in different implementation languages, IRs need to be implemented in more than one programming language. Otherwise compiler components written in different languages cannot interoperate.

Unfortunately, many IRs are only described by one implementation in one language. For these IRs it can be hard to separate the abstract structure of the IR from implementation artifacts. Since IRs are often recursively defined tree-like data structures, once the IR is understood it is easy but tedious to develop different implementations in other languages. Writing functions to pickle the IR is also easy but tedious.

A parser implementation is a poor way to describe the concrete syntax of a programming language. A set of data structures is a poor way to describe an IR. This document describes the Abstract Syntax Description Language (ASDL), a simple declarative language for describing the abstract structures of IRs. IRs described with ASDL are converted into an implementation automatically by tools. Tools generate the data-structure definitions for a target language as well as the pickling functions and other supporting code. ASDL is designed so that it is easy to convert descriptions into readable implementations. ASDL descriptions are more concise than data-structure definitions in languages such as C, C++, and Java.

The idea of special notation for tree-like data structures is not new. Compiler-construction systems and attribute-evaluation tools contain small sublanguages that are descriptions of tree-like data structures [GHL+92, Bat96, Vol91, JPJ+90]. Programming languages that provide support for algebraic data types also have concise notation for defining tree-like data struc-

[1] Zephyr is a compiler infrastructure project conducted jointly at the University of Virginia and Princeton University (see `http://www.cs.virginia.edu/zephyr`). This work is supported in part by the Department of Defense under contract MDA904-97-C-0247 and DARPA order number E381, and the National Science Foundation grant ASC-9612756.

	Good Notation	Language Independent	Simple
Compiler Construction Systems	yes	no	yes
Algebraic Data Types	yes	no	yes
ASN.1	could be better	yes	no
SGML	no	yes	no
Network IDLs	no	yes	yes

Table 1: Evaluation of existing systems

tures [BMS80]. Unfortunately these systems and languages do not solve the problem of providing IR implementations for more than one programming language.

Languages like SGML [GR90] and ASN.1 [ISO87, ITU95b] are not much more than complex description languages for tree-like data structures. These descriptions are declarative specifications of structured data, independent of a particular implementation language. However, these languages have many features not needed in the description of compiler IRs. For example, ASN.1 contains thirteen different primitive string types. SGML has a "tag minimization" feature to help define formats that are easier to write by humans but more difficult to parse.

Although SGML and ASN.1 do solve the problems of component interoperation, they seem verbose, cryptic, and complex. The extra complexity of these systems alone makes them unsuitable, since the resources spent understanding and using the systems can often be greater than the time the systems saves the programmer. There already are groups advocating the use of a simplified subset of SGML for for distribution of web content [XML97]. ASDL in some ways can be viewed as simplification of ASN.1.

Heterogeneous networked systems have solved a similar component interoperation problem with interface description languages (IDLs) that describe abstract interfaces to network services. Tools automatically generate glue code from an IDL description and export a service to the network. ONC RPC [Sri95], OMG CORBA [Obj95], and Xerox's ILU [Xer96] are examples of this approach. Unfortunately CORBA and the other IDLs have awkward encodings for the tree-like data structures seen in IRs.

Ideally those in the compiler research community could reuse existing solutions. Unfortunately none of the existing systems are a good solution to the problem of building interoperable compiler components and concisely defining abstract IRs. Table 1 summarizes our evaluation of existing systems. The following is a summary of the concrete design goals of ASDL.

- The language must be simple and concise.
- The language must be able to encode existing IR's such as SUIF [W$^+$94], FLINT [Sha97], and lcc's IR [FH95].
- Tools that use the language must initially be able to produce code for C, C++, Java, and ML.
- Tools must be able to produce code designed to be understood by programmers, not just other tools.
- Language features must have a natural encoding in all the target languages.

The next sections present the language informally, evaluate the language, and discuss related and future work.

2 ASDL by Example

$$
\begin{array}{rcl}
\mathit{definitions} & = & \{\mathit{typ\_id}\ \texttt{"="}\ \mathit{type}\} \\
\mathit{type} & = & \mathit{sum\_type}\ \mid \mathit{product\_type} \\
\mathit{product\_type} & = & \mathit{fields} \\
\mathit{sum\_type} & = & \mathit{constructor}\ \{\texttt{"}\mid\texttt{"}\ \mathit{constructor}\} \\
 & & [\texttt{"attributes"}\ \mathit{fields}] \\
\mathit{constructor} & = & \mathit{con\_id}\ [\mathit{fields}] \\
\mathit{fields} & = & \texttt{"("}\ \{\mathit{field}\ \texttt{","}\}\ \mathit{field}\ \texttt{")"} \\
\mathit{field} & = & \mathit{typ\_id}\ [\texttt{"?"}\ \mid \texttt{"*"}]\ [\mathit{id}]
\end{array}
$$

Figure 1: Grammar of ASDL. Braces indicate zero or more. Brackets indicate zero or one.

Figure 1 is the grammar for ASDL. The syntax of ASDL has been designed to be natural and intuitively obvious to anyone familiar with context free grammars (CFG) or algebraic data types. In the same way that an unambiguous CFG can be viewed as describing the structure of parse trees, ASDL describes the structure of tree-like data structures. An ASDL description consists of a sequence of productions, which define the structure of a tree-like data structure. ASDL descriptions are tree grammars.

ASDL is simple enough to describe with a few examples. Since one goal is to generate human readable code, there are a few restrictions on ASDL whose rationale is not completely obvious. These restrictions and their motivation are described as they arise. Figure 3 is the ASDL description of a trivial programming language.

2.1 Lexical Issues

$$
\begin{array}{rcl}
\textit{upper} & = & \texttt{"A"} \mid \texttt{...} \mid \texttt{"Z"} \\
\textit{lower} & = & \texttt{"a"} \mid \texttt{...} \mid \texttt{"z"} \\
\textit{alpha} & = & \texttt{"\_"} \mid \textit{upper} \mid \textit{lower} \\
\textit{alpha\_num} & = & \textit{alpha} \mid \texttt{"0"} \mid \texttt{...} \mid \texttt{"9"} \\
\textit{typ\_id} & = & \textit{lower}\ \{\textit{alpha\_num}\} \\
\textit{con\_id} & = & \textit{upper}\ \{\textit{alpha\_num}\} \\
\textit{id} & = & \textit{typ\_id} \mid \textit{con\_id}
\end{array}
$$

Figure 2: Lexical structure

Figure 2 is a description of the lexical structure of tokens used in the ASDL grammar in Figure 1. The names of constructors and types in the description contain informal semantic information that should be preserved by a tool when translating descriptions into implementations. To keep the mapping from ASDL names to target language names simple, the names of types and constructors are restricted to the intersection of valid identifiers in the initial set of target languages. To help the reader distinguish between types and constructor names, types are required to begin with a lower case letter and constructor names must begin with an upper case letter. Rather than restricting ASDL names to exclude the union of keywords in all target language, ASDL tools will have to keep track and correct conflicts between target language keywords and the type and constructor names.

ASN.1 has a similar restrictions. However, the ASN.1 equivalent of ASDL types must begin with an upper case letter, and non-type identifiers must begin with a lower case letter. The ASN.1 restrictions are incompatible with many common stylistic conventions in ML, Java, C++, and C. For example, enumerated constants in ASN.1 must begin with a lower case letter, but C style languages conventionally use all uppercase identifiers for enumerated constants.

2.2 ASDL Fundamentals

An ASDL description consists of three fundamental constructs: types, constructors, and productions. A type is defined by productions that enumerate the constructors for that type. In Figure 3 the first production describes a *stm* type. A value of the *stm* type is created by one of three different constructors **Compound**, **Assign**, and **Print**. Each of these constructors has a sequence of

$$
\begin{array}{rcl}
\textit{stm} & = & \textbf{Compound}(\textit{stm}, \textit{stm}) \\
 & \mid & \textbf{Assign}(\textit{identifier}, \textit{exp}) \\
 & \mid & \textbf{Print}(\textit{exp\_list}) \\
\textit{exp\_list} & = & \textbf{ExpList}(\textit{exp}, \textit{exp\_list}) \mid \textbf{Nil} \\
\textit{exp} & = & \textbf{Id}(\textit{identifier}) \\
 & \mid & \textbf{Num}(\textit{int}) \\
 & \mid & \textbf{Op}(\textit{exp}, \textit{binop}, \textit{exp}) \\
\textit{binop} & = & \textbf{Plus} \mid \textbf{Minus} \mid \textbf{Times} \mid \textbf{Div}
\end{array}
$$

Figure 3: Simple ASDL description

fields that describe the type of values associated with a constructor.

The **Compound** constructor has two fields whose values are of type *stm*. One can interpret the production as defining the structure of *stm* trees which can have three different kinds of nodes **Compound**, **Assign**, and **Print** where the **Compound** node has two children that are subtrees that have the structure of a *stm* tree.

Notice that the *binop* type consists of only constructors which have no fields. Types like *binop* are therefore finite enumerations of values. Tools can easily recognize this and represent these types as enumerations in the target language. ASDL does not provide an explicit enumeration type, unlike ASN.1 and the various IDLs. Tools should recognize this idiom and use an appropriate encoding.

There are three primitive pre-defined types in ASDL. Figure 3 uses two of them *int* and *identifier*. The *int* type represents signed integers of infinite precision. Specific tools may choose to produce language interfaces that represent them as integers of finite precision. These language interfaces should appropriately signal an error when they are unable to represent such a value during unpickling. The *identifier* type is analogous to Lisp symbols. ASDL also provides a primitive *string* type.

2.3 Generating Code from ASDL Descriptions

From the definitions in Figure 3, it is easy to automatically generate data type declarations in target languages such as C, C++, Java, and ML. For languages like C, each type is represented as a tagged union of values. Languages like Java and C++ have a single abstract base class for each type and concrete subclasses of the base class for each variant of the type.

Figure 4 shows one way to translate the *stm* type into C. Each ASDL type is represented as a pointer to a structure. The structure contains a “kind” tag that indicates which variant of the union the current value holds. It is also convenient to have functions that allocate space and properly initialize the different variants of *stm*. Notice that the *binop* is translated as an enumeration.

```
typedef struct _stm *stm_ty;
struct _stm {
   enum {Compound_kind=1, Assign_kind=2,
         Print_kind=3} kind;
   union {
      struct { stm_ty stm1; stm_ty stm2; } Compound;
      struct { ...} Assign;
      struct { ...} Print;
   } v;
};
...
enum binop_ty {Plus=1, Minus=2, Times=3, Div=4};
...
stm_ty Compound (stm_ty stm1, stm_ty stm2) {
   stm_ty p;

   p = malloc(sizeof(*p));
   p->kind = Compound_kind;
   p->v.Compound.stm1 = stm1;
   p->v.Compound.stm2 = stm2;
   return p;
}
stm_ty Assign (identifier_ty identifier1, exp_ty exp1) { ...}
stm_ty Print (exp_list_ty exp_list1) { ...}
...
```

Figure 4: Simple translation to C

```
abstract public class stm {
   protected int _k;
   public final int kind(){ return _k; }
   public static final int Compound = 1;
   public static final int Assign = 2;
   public static final int Print = 3;
}

public class Compound extends stm {
   public stm stm1; public stm stm2;
   public Compound(stm stm1, stm stm2) { ...}
}

public class Assign extends stm { ...}
public class Print extends stm { ...}
...
```

Figure 5: Simple translation to Java

```
datatype stm = Compound of (stm * stm)
             | Assign of (identifier * exp)
             | Print of (exp_list)
             ...
```

Figure 6: Simple translation to ML

Figure 5 shows one possible encoding in Java, which is applicable for many other languages with objects, such as C++ and Modula-3. The *stm* type is represented as an abstract base class. The various ASDL constructors are translated into subclasses of the abstract base class. Each constructor class inherits a tag that identifies which variant of *stm* it is. The translation to a language like ML that has algebraic data types is almost trivial (See Figure 6).

Our prototype definitions generator tool uses these encoding schemes to automatically translate ASDL into C, C++, Java, and ML. These encodings are simple and uniform; they are not necessarily the most efficient possible. Better tools can potentially generate more efficient encodings, or allow the programmer to specify an encoding explicitly.

2.4 Field Names

Since languages like C, Java, and C++ access components of aggregates with named fields, ASDL descriptions allow the specification of a field name to access the values of constructor fields. In the absence of a supplied field name tools can easily create field names based on the position and type of a constructor field. Since field names often encode semantic information, the ability to provide names for fields in the descriptions improves the readability of descriptions and the code generated from those descriptions. There are no restrictions on the case of the first character of field names. Figure 7 contains a fragment of the original description which also includes field names.

2.5 Sequences

The *exp_list* type illustrates a common idiom for expressing a uniform sequence of some type. Sequences of a uniform type occur throughout descriptions and general programming. ASDL provides special support for sequences of values through the "*" (sequence) qualifier, which means that the type of some value is an sequence of zero or more elements of that type. Figure 8 demonstrates its use in the context of the previous definitions. The sequence qualifier is not just syntactic sugar. It provides a mechanism in the description for the writer to more clearly specify the intent giving tools that generate code more freedom to use appropriate representations in the native language. For example, a tool may translate a

$$\begin{array}{rcl} \textit{stm} & = & \textbf{Compound}(\textit{stm}\ \text{head},\ \textit{stm}\ \text{next}) \\ & | & \textbf{Assign}(\textit{identifier}\ \text{lval},\ \textit{exp}\ \text{rval}) \\ & | & \textbf{Print}(\textit{exp\_list}\ \text{args}) \\ \textit{exp\_list} & = & \textbf{ExpList}(\textit{exp}\ \text{head},\ \textit{exp\_list}\ \text{next}) \\ & | & \textbf{Nil} \\ & \ldots & \end{array}$$

Figure 7: ASDL description with named fields

$$\begin{array}{rcl} \textit{stm} & = & \textbf{Compound}(\textit{stm}\ \text{head},\ \textit{stm}\ \text{next}) \\ & | & \textbf{Assign}(\textit{identifier}\ \text{lval},\ \textit{exp}\ \text{rval}) \\ & | & \textbf{Print}(\textit{exp}^*\ \text{args}) \\ & \ldots & \end{array}$$

Figure 8: ASDL description with sequences

sequence type into an array or another built-in sequence type that the target language supports, such as a polymorphic or templated list type.

ASN.1, SGML, ONC RPC, OMG IDL, and Xerox's ILU have qualifiers for sequence types. These systems, except for ASN.1 and SGML, also have qualifiers to specify the minimum or maximum length of a sequence. ASN.1 and SGML also have qualifiers to specify that the order of components in sequences has no meaning. They support the notion of a set of values. ASDL does not support this feature, since sequences can model sets.

2.6 Product Types, Attributes, and Options

Those familiar with EBNF[Wir77] or algebraic data types may expect to be able to write descriptions with productions such as

$$t \quad = \quad \textbf{C}(\textit{int},\ (\textit{int},\ \textit{int})^*)\ .$$

However, complex expressions of this type are not allowed in ASDL. The reason behind this restriction is that not all the source languages support a natural encoding for complex type expressions. One would expect that equivalent type expressions are translated into compatible types in the target language. Since the semantics of aggregate types in C and C++ require each new aggregate (struct/class) definition to be a new distinct type, tools would have to use target language type abbreviation mechanisms (e.g. typedef) to achieve this effect. So a tool must assign a name to the type that a programmer must use and remember. There are several obvious ways to automatically generate type names for the expressions, however it would be preferable to require description writers to provide semantically meaningful names to these intermediate types, since generated code is intended to be readable by the programmer. So the above would be written as

$$\begin{array}{rcl} t & = & \textbf{C}(\textit{int},\ \textit{int\_pair}^*) \\ \textit{int\_pair} & = & \textbf{IP}(\textit{int},\ \textit{int})\ . \end{array}$$

This restriction is unsatisfactory since it requires descriptions writers to also provide a name for the single constructor, **IP**, of this type. To overcome this problem, ASDL provides (Cartesian) product types which are productions that define a type that is an aggregate of several values of different types. Product types are also restricted in that they can not lead to recursive definitions, since recursive product type definitions to not describe tree structures. Another way to encode the first expression in ASDL which avoids the extra constructor would be

$$\begin{array}{rcl} t & = & \textbf{C}(\textit{int},\ \textit{int\_pair}^*) \\ \textit{int\_pair} & = & (\textit{int},\ \textit{int})\ . \end{array}$$

Often several constructors of a type share a set of common values. To make this explicit, ASDL includes an attribute notation. Since all variants of a type carry the values of an attribute, its fields can be accessed without having to discriminate between the various constructors. Attributes can be seen as providing some limited features of inheritance.

Most languages provide the notion of a special distinguished empty value (NULL, nil, NONE). ASDL provides a convention for specifying that certain values may be empty with the "?" (optional) qualifier.

$$\begin{array}{rcl} \textit{pos} & = & (\textit{string}?\ \text{file},\ \textit{int}\ \text{line},\ \textit{int}\ \text{offset}) \\ \textit{stm} & = & \textbf{Compound}(\textit{stm}\ \text{head},\ \textit{stm}\ \text{next}) \\ & | & \textbf{Assign}(\textit{identifier}\ \text{lval},\ \textit{exp}\ \text{rval}) \\ & | & \textbf{Print}(\textit{exp}^*\ \text{args}) \\ & & \text{attributes}\ (\textit{pos}\ \text{p}) \\ \textit{real} & = & (\textit{int}\ \text{mantissa},\ \textit{int}\ \text{exp}) \\ \textit{exp} & = & \textbf{Id}(\textit{identifier}) \\ & | & \textbf{Num}(\textit{int}) \\ & | & \textbf{Op}(\textit{exp},\ \textit{binop},\ \textit{exp}) \\ & & \text{attributes}\ (\textit{real}?\ \text{value}) \\ \textit{binop} & = & \textbf{Plus}\ |\ \textbf{Minus}\ |\ \textbf{Times}\ |\ \textbf{Div} \end{array}$$

Figure 9: ASDL description with products, attributes, and options

Figure 9 is an example of an ASDL description that uses, products, attributes, and options. It is important to emphasize that ASDL says nothing about how a definition should be translated by a tool into a specific concrete implementation. The description language and external data encoding are fixed; the particular target language interfaces are not. Different tools may produce different language interfaces, as long as the pickle formats used by various tools are compatible. For example since ASDL does not provide a primitive type for real numbers the ASDL description in Figure 9 describes a real type in terms of two arbitrary precision integers. Programmers can provide the translation tool hints and conversion functions so actual implementations of the above IR use native floating-point values.

ASDL product types are nothing more than records. Many IDLs (XDR, ISL, and IDL) which have support for records place a similar restriction on the recursive definitions of structures. The description language for Xerox's ILU system (ISL) forbids complex type expressions in the same way ASDL does. The XDR specification allows for complex type expressions, but common implementations of the tools do not allow the use of them.[2] ASN.1, SGML, and OMG's IDL allow the construction of complex type expressions. All the previously mentioned languages have a similar optional qualifier. Although attributes can be simulated in all the description languages, only SGML has a notion of attributes similar to ASDL.

2.7 Pickles

```
...
void pkl_write_exp( ... ) { ... }
void pkl_write_exp_list( ... ) { ... }
void pkl_write_binop( ... ) { ... }
void pkl_write_stm(stm_ty x, outstream_ty s) {
  switch(x->kind) {
    case Compound_kind:
      pkl_write_int(1, s);
      pkl_write_stm(x->v.Compound.stm1, s);
      pkl_write_stm(x->v.Compound.stm2, s);
      break;
    case Assign_kind:
      pkl_write_int(2, s);
      ...
      break;
    case Print_kind:
      pkl_write_int(3, s);
      ...
      break;
    default: pkl_die();
  }
}
...
exp_ty pkl_read_exp(instream_ty s) { ... }
...
stm_ty pkl_read_stm(instream_ty s) { ... }
```

Figure 10: Automatically Generated Pickler

[2] `rpcgen` on Solaris and OSF 3.2

Since ASDL data structures have a tree-like form, they can be represented linearly with a simple prefix encoding. It is easy to generate functions that convert to and from the linear form. Figure 10 is a generated routine that "pickles" the *stm* type seen previously in Figure 4. A pre-order walk of the data structure is sufficient to convert a *stm* to its pickled form. The walk is implemented as recursively defined functions for each type in an ASDL definition. Each function visits a node of that type and recursively walks the rest of the tree.

In Figure 10 the function `pkl_write_stm` dispatches based on the kind of *stm* constructor of the node being visited. It visits the node by writing a unique tag to identify the constructor to an output stream and then recursively visits any values carried by the constructor. Tags are assigned based on the order of constructor definition in the description. Values are visited from left to right based of the order in the definition. If there are any attribute values associated with a type, they are visited in left to right order after writing the tag but before visiting the values unique to a given constructor. In this case there are no attribute values. Since the prefix encoding does not represent pointers in the data structure the linear form is significantly smaller than the pointer data structures.

The function `pkl_write_stm` calls `pkl_write_int` to output integer values to the output stream. Since ASDL integers are intended to be of infinite precision they are represented with a variable-length, signed-magnitude encoding. If most integer values tend to be values near zero, this encoding of integers may use less space than a fixed precision representation.

Sequence types are represented with an integer length-header followed by that many values. Optional values are preceded by an integer header that is either one or zero. A zero indicates that the value is empty (NONE, nil, or NULL) and no more data follows. A one indicates that the next value is the value of the optional value. Identifiers and strings are encoded with an integer size-header followed by the raw bytes needed to reconstruct the string or identifier. All the headers are encoded with the same arbitrary precision integer encoding described previously.

Product types are written out sequentially without any tag. 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 there is no explicit type information in the pickle. The prefix encoding of trees, variable-length integer encoding, and lack of explicit type information, all help keep the size of pickles small. Smaller pickles reduce the system IO since there is less data to write or read. Smaller pickles are also more likely to fit completely in the cache of the IO system.

The ASDL pickle format resembles the Packed Encoding Rules (PER) of ASN.1 [ITU95a]. Like the ASDL pickle format the PER is a prefix encoding of tree values. Neither format encodes redundant type information. Rather than using a variable-precision encoding for integer values and headers, the PER determines from the ASN.1 specification the maximum precision need for a particular value and uses fixed precision integers to represent those values. In the case where an ASN.1 specification does not constrain an integer value so the maximum precision can be determined, ASN.1 resorts to a variable precision integer encoding. The PER encoding of optional values is also slightly different from the ASDL approach. PER optional values are encoded as a bitmap that precedes a record of values that may contain optional values.

Preliminary performance evaluations of the generated pickled code suggest that they are efficient enough not to be the primary performance bottlenecks. Writing pickled values is dominated by IO time, while reading values is dominated by memory allocator time.

3 Evaluation

The next few sections describe insights gained by attempting to respecify an existing compiler IR in ASDL, an evaluation of ASDL's syntax, and some initial experiences using ASDL related tools to build applications.

3.1 ASDL SUIF

ASDL has been used to respecify the core IR of an existing compiler infrastructure, the Stanford University Intermediate Format [W+94] (SUIF) written in C++. Being able to specify existing compiler IRs in ASDL is one of the key design goals of ASDL. SUIF uses an object oriented framework to implement its core IR.

$$
\begin{array}{rcl}
\textit{instruction} & = & \textbf{In\_rrr}(\ldots) \\
 & | & \textbf{In\_ldc}(\ldots) \\
 & & \ldots \\
 & | & \textbf{In\_gen}(\ldots) \\
 & & \textsf{attributes}\ (\ldots)
\end{array}
$$

Figure 12: ASDL encoding of SUIF class hierarchy

Figure 11 shows the class hierarchy for SUIF. Looking at classes such as *sym_node* and *instruction*, it is easy to model these classes as types in ASDL with their subclasses represented as constructors in ASDL. Fields that the subclasses may inherit from *instruction* are modeled as common fields and use the attribute mechanism in ASDL. Figure 12 outlines this approach for the *instruction* class.

There are situations where the intent of C++ SUIF code does not fit well into the ASDL model, but these situations are isolated. For example *proc_symtab*, *tree_proc*, and *enum_type* require us to simulate two levels of inheritance in the ASDL description. Attributes in ASDL provide only one level of inheritance. To handle cases where class hierarchies have more than one level of inheritance, extra intermediate types have to be introduced, making the ASDL description less than perfect.

Figure 13 demonstrates how to simulate two levels of inheritance in ASDL. The class *tree_proc* inherits from *tree_block*. The class *tree_block* inherits from *tree_node*. The ASDL description models this by introducing a new intermediate type *tree_block_rest* which consists of two constructors. The **Tree_block** constructor has all the fields from the *tree_block* class. The **Tree_proc** constructor contains or all the fields inherited from the *tree_block* class and the fields of *tree_proc* class. There is a slightly better encoding that uses attributes, since the **Tree_block** and **Tree_proc** constructors share a common set of fields.

The C++ code also uses subtyping to express the constraint that a field must be a particular subtype of an abstract class. In ASDL this is equivalent to using an ASDL constructor as a type in the description. This problem can be solved by allowing the ASDL encoding to be more permissive by not encoding this constraint. Figure 14 provides an example. The `pr` field is declared as a *tree_proc* class, which is a subtype of *tree_node*. In the ASDL description the *tree_proc* corresponds to a constructor, not a type, so `pr` in the ASDL description cannot be declared with the proper type. Instead the ASDL description must use the *tree_node* type.

There are issues not unique to C++, such as how to encode pointers to other tree nodes, making the data structures arbitrary graphs, or the encoding of pointers to other external data structures such as symbol tables. These issues can easily be handled by including unique identifiers and an auxiliary mapping from identifiers to values, to simulate the effect of pointers.

Although the ASDL description is not a verbatim translation of the C++ implementation, the majority of the ASDL description captures most of the features of the C++ implementation in a natural way. The ASDL encoding is less restrictive than the C++ implementation, so functions can be written that convert between data structures that use the original C++ implementation of SUIF and the equivalent ASDL data structures and without loss of information. Using these functions along with code automatically generated from the ASDL description, we have built a tool tool that allows the compilers written in ML and Java to interface to the existing

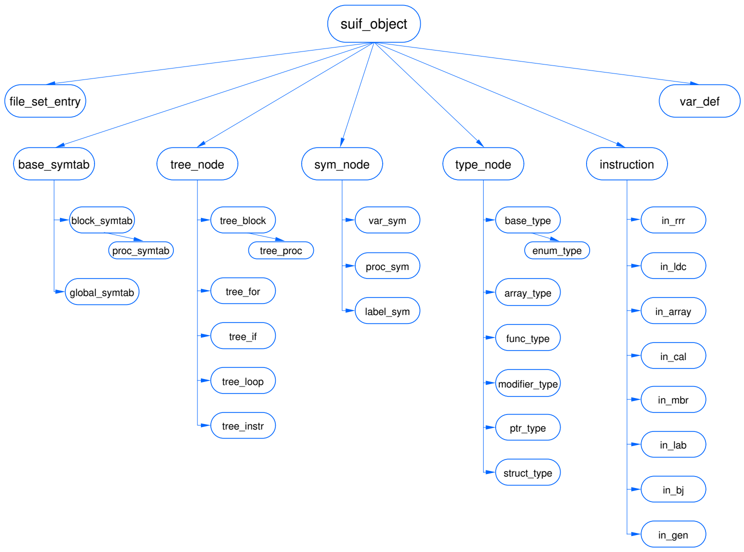

Figure 11: SUIF class hierarchy

SUIF compiler components[Ser97].

3.2 ASDL Syntax

Table 2 compares the size of ASDL SUIF description and the C++ implementation. The C++ kernel is the core set of source files that defines the structure of the SUIF IR and the related support functions. From the kernel there are ten core header files that describe IR structure. The ASDL description was written by examining the original C++ sources. The ASDL description uses the same set of identifiers that the original C++ code uses. The ASDL description does not completely capture all aspects of the C++ code as, explained previously. Table 2 reports the total numbers of lines, words, and characters as reported by the UNIX `wc` command for each set of these files. It is clear that ASDL description is more compact than the C++ implementation.

A qualitative comparison between the ASDL and alternative systems syntax can be found in Appendix A. It contains an ASDL description and semantically equivalent encoding of the description in various other specification languages (ASN.1, SGML, GMD's ast, and OMG's IDL). Of the specification languages, the ASN.1 specification seems to be comparable in clarity to ASDL. Though syntax is sometimes a matter of taste, the lexical restrictions of ASN.1 makes translation of an ASN.1 identifiers into idiomatic target language identifiers more complicated than necessary.

3.3 ASDL Tools

We have constructed the following tools:

- A prototype definitions generator that reads ASDL descriptions and produces data structures defintions and pickling routines in C, C++, Java, and ML
- A browser-editor that can graphically view and manipulate arbitrary ASDL pickles
- A tool to convert between the original C++ SUIF data structures and data structures produced from the ASDL description by the definitions generator[Ser97].

3.3.1 Prototype Definitions Generator

ASDL has been used to describe the internal data structures of a prototype tool that generates code from ASDL descriptions. Rather than manipulating raw strings, the tool works with data structures that represent the abstract syntax trees (AST) of the target languages (C, C++, Java, ML). The AST is then pretty printed [Opp80] to produce the final output. The tool uses the translation techniques outlined in Section 3. A related tool produces a set of C++ functions that automatically pickle and unpickle the C++ data structures.

During the initial design process of ASDL the first thing discussed was the abstract syntax of ASDL. A proposal for the abstract syntax of ASDL was written using the algebraic data type notation of ML. A purposed concrete syntax was also discussed along with the abstract syntax. The clean separation between the abstract and concrete syntax helped isolate important issues of language design from issues of syntax. Although the initial concrete syntax was substantially modified for the

Original C++ code

class *tree_node*		...
class *tree_block*	:	public *tree_node* $\{ty_1\ \mathrm{f}_1; \ldots; ty_n\ \mathrm{f}_n;\}$
class *tree_proc*	:	public *tree_block* $\{ty'_1\ \mathrm{f}'_1; \ldots; ty'_n\ \mathrm{f}'_n;\}$

Encoding without Attributes

tree_node	=	...
	|	**Tree_block_rest**(*tree_block_rest*)
tree_block_rest	=	**Tree_block**$(ty_1\ \mathrm{f}_1, \ldots, ty_n\ \mathrm{f}_n)$
	|	**Tree_proc**$(ty_1\ \mathrm{f}_1, \ldots, ty_n\ \mathrm{f}_n, ty'_1\ \mathrm{f}'_1, \ldots, ty'_n\ \mathrm{f}'_n)$

Encoding with Attributes

tree_node	=	...
	|	**Tree_block_rest**(*tree_block_rest*)
tree_block_rest	=	**Tree_block**
	|	**Tree_proc**$(ty'_1\ \mathrm{f}'_1, \ldots, ty'_n\ \mathrm{f}'_n)$
		attributes$(ty_1\ \mathrm{f}_1, \ldots, ty_n\ \mathrm{f}_n)$

Figure 13: Encoding inheritance in ASDL

class *sym_node*	:	...
class *proc_sym*	:	public *sym_node* { ... *tree_proc* *pr; ... }
sym_node	=	**Proc_sym**(..., **tree_node** pr,...)

Figure 14: Ignoring subtyping constraints

asdl_ty	=	**Sum**(*identifier*, *field**, *constructor*, *constructor**)
	|	**Product**(*identifier*, *field*, *field**)
constructor	=	**Con**(*identifier*, *field**)
field	=	**Id**(*identifier*, *identifier*?)
	|	**Option**(*identifier*, *identifier*?)
	|	**Sequence**(*identifier*, *identifier*?)

Figure 15: Abstract Syntax of ASDL in ASDL

current version of ASDL syntax, the abstract syntax has changed little from the initial proposal.

The abstract syntax for ASDL can itself be expressed in ASDL (see Figure 15). This is an important property that is used by the browser-editor. Since an ASDL description can be represented as an ASDL value, after a parser converts the concrete syntax of an ASDL description into its abstract form the description can itself be pickled. Other tools can read and manipulate the abstract syntax without any dependence on the concrete syntax. The browser is one such tool.

3.3.2 Graphical Pickle Browser and Editor

The browser is a graphical tool for viewing and editing arbitrary pickled ASDL values. The browser reads in two pickles to do this. One is an arbitrary pickled ASDL value. The other it the pickled ASDL description that contains all the ASDL types that occur in the first pickle. Given the ASDL description for the first pickle the brower-editor is able to display the first pickle as a hierarchical list or a graphical tree. It allows the user to specify how each kind of node is drawn by allowing the selection of colors, fonts, etc. Trees can be edited using standard cut and paste operations or by creating/modifying nodes. If the user double-clicks on a particular constructor, a pop-up menu will appear allowing the user to change which constructor-type the node should have. Upon selecting a type for a node, the browser fills in new nodes for the children of that type automatically. Pickles edited with the browser can be saved as a new pickled value.

The browser is written in C. It manipulates a C version of the ASDL abstract syntax produced by the definition generator from the ASDL description of ASDL. When the user edits an object, the browser modifies an abstract representation of generic ASDL values in mem-

	files	lines	words	characters
C++ kernel (with comments)	63	25,095	76,094	632,693
C++ core files (without comments)	10	2,316	6,533	60,984
ASDL description	1	204	562	6,921

Table 2: Code size comparison of SUIF data structures.

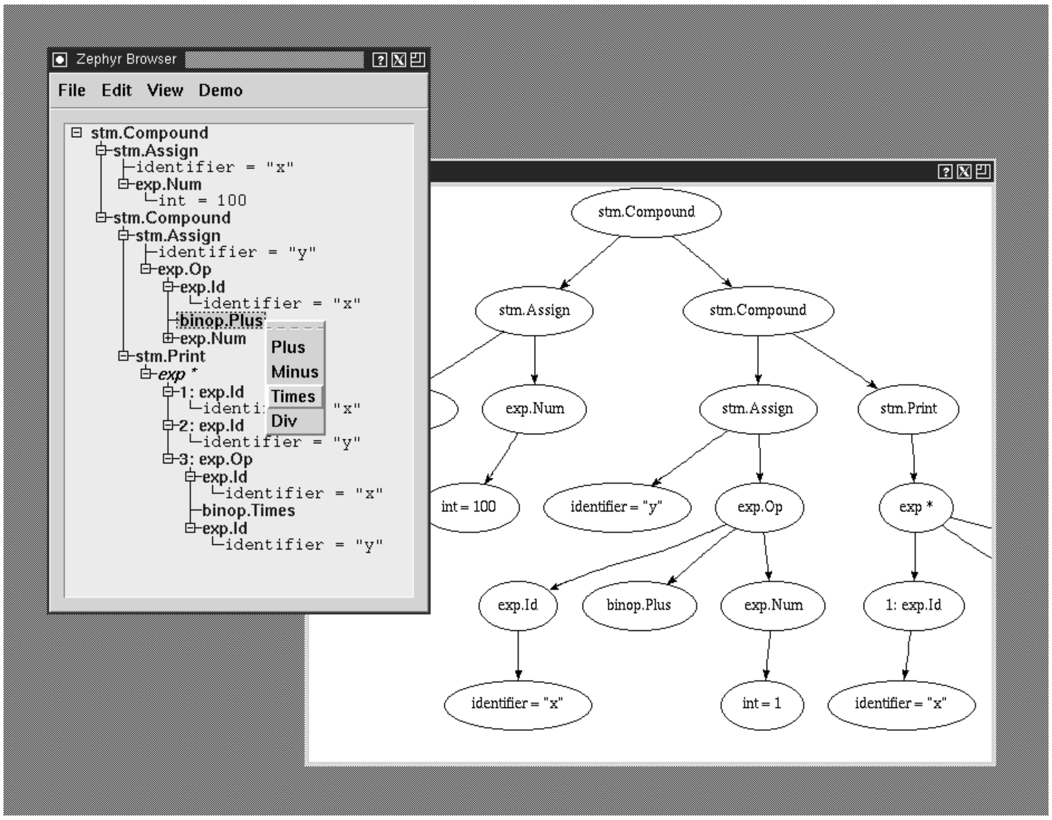

Figure 16: Browser-editor for ASDL pickles

value	=	**SumVal**(*identifier*, *value**, *value**)
	|	**ProductVal**(*value*, *value**)
	|	**SequenceVal**(*value**)
	|	**NoneVal**
	|	**SomeVal**(*value*)
	|	**PrimVal**(*priml*)
prim	=	**IntVal**(*int*)
	|	**IdentifierVal**(*identifier*)
	|	**StringVal**(*string*)

Figure 17: ASDL values specified in ASDL

ory, also generated from the ASDL description in Figure 17. The in memory representation is then converted into the correct pickle format. This makes the browser completely independent of the actual details of both the concrete syntax of ASDL or the actual pickling format. As long as the abstract structure of the pickles and concrete syntax stay unchanged, changes to the details of the pickle format or concrete syntax have little impact on the browser. Since ASDL descriptions are also pickled values the browser can be used to create, edit, or view ASDL type descriptions by manipulating the abstract syntax of ASDL directly. The browser-editor and parser for the concrete syntax of ASDL are two different user interfaces to building the abstract syntax of ASDL descriptions.

Since the ASDL approach allows applications written in different programming languages to interoperate, the browser is implemented in C allowing the use of an existing and advanced GUI toolkit such as Tcl/Tk. The definitions generator and the parser for ASDL descriptions are implemented in Standard ML, but these two tools easily interoperate with each other because of the standard pickle format. ASDL gives developers more flexibility in choosing the appropriate language for a given task.

4 Related Work

Automatically encapsulating runtime data structures into external out of core values is not an idea unique to ASDL. Some languages, notably Modula-3, have built-in language support for translating values into "pickles". Unlike ASDL, Modula-3 pickling is able to handle arbitrary graph structures and does pickling based on runtime type information and support from a garbage collector. ASDL pickling is based on compile time static information, which makes it less flexible than Modula-3 but more portable and efficent. ASDL pickling code need not depend on any special runtime support and can be optimized based on static information. Java has borrowed the pickling techniques and ideas from Modula-3 to provide the automatic "serializing" of arbitrary language objects.

Pizza [OW97] is a superset of Java that provides algebraic data types, giving Java concise notation for tree-like data structures. A combination of Pizza along with the automatic serialization of types in Java has some similarities to the ASDL approach. Unfortunately the pickles that Java produces are not meant to be language independent.

The most similar work to ASDL is ASN.1. Ignoring syntactic issues, ASDL, resembles a subset of ASN.1. The original evaluation of the existing systems suggests that ASN.1 can solve the component interoperation problem. Commercial tools exist that translate ASN.1 into C, C++, and Java, so it is tempting to use ASN.1 and write the remaining tools for languages not supported by commercial systems. However, ASDL has one significant advantage over ASN.1. ASDL is much simpler than the full ASN.1 specification language. Simplicity is not just an esthetic concern. The complexity of ASN.1 makes it difficult to write tools that use it. Considering ASDL is able to solve the problems in the compiler domain, the extra effort in dealing with the complexity of ASN.1 complexity does not seem worth the effort.

The official context free grammar of the most recent version of ASN.1 contains over 150 non-terminals and 300 productions, and occupies eight pages [ITU95b]. The equivalent ASDL context free grammar contains a little over ten non-terminals, twenty productions, and easily fits on half a page. The size of the ASN.1 grammar alone makes it difficult to build tools for it. A freely available ASN.1 compiler [Sam93], which converts a subset of ASN.1 to C and C++ but parses the full ASN.1 language, has a 3000 line `yacc` grammar. The rest of the system consists of 13000 lines of C. Because of ASDL's simplicity, it is easy to construct a definitions generator for different languages. The prototype tool described in this document took a few weeks to implement and is around 5000 lines of ML code. This prototype generates definitions for four different languages.

Although ASN.1 is intended to describe network data, it is also used to describe data in other domains, such as chemical abstracts [CXF94] and gene sequences [NCB96]. These ASN.1 specifications exist to help improve the exchange of information across software systems written to manipulate data in these domains. Close inspection of the ASN.1 specifications for these domains reveals that they only use a small subset of the features of ASN.1 and that the subset they use is very close to ASDL. This suggests that ASDL has wider applications, and that it is worthwhile to develop a strategy to interoperate with existing systems using ASN.1.

5 Future Work

As the SUIF encoding demonstrates, realistic ASDL descriptions may still be reasonably long. ASDL should support modularized descriptions. Modularizing descriptions at the ASDL level requires us to address the issue of how modular descriptions are translated into the target language. Should each module of the description correspond to a compilation unit in the target language? Should cyclic module dependencies be allowed? Cyclic module dependencies are convenient when describing ASTs, but languages like ML do not support cyclic module dependencies.

More work needs to be put into building tools that use the ASDL definitions. Our current prototype tool performs a naive translation of an ASDL description into target languages. The tools that generate ASDL descriptions need to have more hooks so that users can control how descriptions are translated. Tools could also perform more aggressive automatic representation optimizations on the generated code.

It seems appropriate to reuse ASDL descriptions for a wide variety of other tools, such as attribute evaluators, parsers, pattern matchers, and pretty printers. All these systems can benefit from the formalisms that ASDL provides. Jansson [JJ97] presents a formalism (polytypic programming) to describe functions that generate functions based on structural induction on an arbitary algebraic data type. Polytypic programming allows the creation of generator generators. Jansson's approach could be extended into a tool that generates code generators for ASDL data types. A polytypic description that inductively describes the equality of arbitary types can be turned automatically into a program that takes an ASDL description and produces another program to check for equality.

6 Conclusions

Declarative languages such as regular expressions and context-free grammars, with tools like `lex` and `yacc`, help popularized the notion of describing the concrete syntax of programming languages formally. Description languages like ASDL will popularize the notion of formally describing the abstract syntax of programming languages and the internal representations of compilers. Our initial experience with SUIF and other descriptions suggests that ASDL is able to encapsulate the fundamental structures of important data structures in a concise and language independent way.

The core idea of concise notation for describing tree like-data structures behind ASDL is so simple it has been reinvented in different guises by several systems that span a variety of domains. ASN.1 is one such system, which also provides support for cross language interoperation. Unfortunately the full ASN.1 language is complex, making tool development a difficult task. ASDL represents a simple and powerful subset of ASN.1. The simplicity of ASDL allows for easier implementation of tools that use it.

7 Availability

ASDL is part of a larger project to develop a resuable compiler infrastructure. The most up to date information on ASDL can be found at `http://www.cs.virginia.edu/zephyr`. We are currently working on a production quality release of the software and documentation, which should be completed in January 1998. Working releases of the software will be available in the interim; see the web page for details.

8 Acknowledgments

We would like to thank all those who have helped out along the way including the developers of SUIF, Norman Ramsey, and David Hanson.

9 Glossary of Acronyms

ASDL Abstract Syntax Description Language

ASN.1 Abstract Syntax Notation One

AST Abstract Syntax Tree

CORBA Common Object Request Broker Architecture

EBNF Extended Backus Naur Form

IDL Interface Description Language

ILU Inter Language Union

IR Intermediate Representation

ISL Interface Specification Language

ONC Open Network Consortium

OMG Object Management Group

PER Packed Encoding Rules

RPC Remote Procedure Call

SGML Standard Generalized Markup Language

SUIF Stanford University Intermediate Format

XDR External Data Representation

References

[Bat96] Rodney M. Bates. Examining the Cocktail toolbox. *Dr. Dobb's Journal of Software Tools*, 21(3):78, 80–82, 95–96, March 1996.

[BMS80] R. Burstall, D. MacQueen, and D. Sannella. Hope: an experimental applicative language. In *Proceedings of the 1980 LISP Conference*, pages 136–43, Stanford, 1980.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 217–230, 1993.

[CXF94] Chemical exchagne format. `ftp://info.cas.org/pub/cxf`, 1994.

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, February 1992.

[GR90] Charles F. Goldfarb and Yuri Rubinsky. *The SGML handbook*. Clarendon Press, Oxford, UK, 1990.

[ISO87] Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1). International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8824.

[ITU95a] Information Technology – Abstract Syntax Notation One (ASN.1): Encoding Rules – Packed Encoding Rules (PER). International Telecommuncation Union, 1995. ITU-T Recommendation X.691.

[ITU95b] Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. International Telecommuncation Union, 1995. ITU-T Recommendation X.680.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 15–17 January 1997.

[JPJ+90] M. Jourdan, D. Parigot, C. Julie, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 209–222, White Plains, New York, June 1990.

[NCB96] National center for biotechnology software development toolkit. `ftp://ncbi.nlm.nih.gov/toolbox/ncbi_tools`, 1996.

[Obj95] Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, 1995.

[Opp80] Dereck C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings POPL 1997*, Paris, January 15-17 1997.

[Sam93] Michael Sample. Snacc 1.1. `http://www.nsg.bc.ca/Software.html`, 1993.

[Ser97] Christopher S. Serra. Bridging suif and zephyr:a compiler infrastructure interchange. Princeton University Senior Thesis, May 1997.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[Sri95] Raj Srinivasan. RFC 1831: RPC: Remote Procedure Call Protocol specification version 2, August 1995.

[Vol91] J. Vollmer. Experiences with gentle: Efficient compiler construction based on logic programming. *Lecture Notes in Computer Science*, 528:425–??, 1991.

[W+94] Robert Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[Wir77] N. Wirth. What can be do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):882, November 1977.

[Xer96] Xerox Corporation. *ILU 2.0alpha8 Refernce Manual*, May 1996. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[XML97] Extensible markup language (XML). `http://www.w3.org/TR/WD-xml`, 1997.

A Appendix A

A.1 Zephyr ASDL

```
stm = Compound(stm head, stm next)
    | Assign(identifier id, exp exp)
    | Print(exp* args)
exp = Id(identifier id)
    | Num(int v)
    | Op(exp lval, binop bop, exp rval)
binop = Plus | Minus | Times | Div
```

A.2 GMD's compiler toolkit

```
-- See http://www.gmd.de/SCAI/lab/adaptor/ast.html
Stm = <
   Compound = head: Stm next: Stm .
   Assign   = [id: char*] exp: Exp .
   Print    = args: ExpList .
> .
Exp = <
   Id  = [id: char*] .
   Num = [v: int] .
   Op  = lval: Exp bop: Binop rval: Exp .
> .
Binop = < Plus = .
          Minus = .
          Times = .
          Div   = .
        > .
```

A.3 ISO X.680 ASN.1

```
-- See http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x680_27252.html
Stm ::= CHOICE {
         compound SEQUENCE {head Stm, next Stm},
         assign SEQUENCE {head Stm, next Stm},
         print SEQUENCE {args SEQUENCE OF Exp}
        }
Exp ::= CHOICE {
         id STRING,
         num INTEGER,
         op SEQUENCE {lval Exp,bop BinOp,rval Exp}
        }
Binop ::= ENUMERATED {plus, minus, times, div}
```

A.4 SGML DTD

```
<!ENTITY % id  "(#PCDATA)">
<!ENTITY % int  "(#PCDATA)">
<!ENTITY % binop "(Plus|Minus|Times|Div)">
<!ENTITY % stm  "(Compound|Assign|Print)">

<!ELEMENT Compound - - (%stm,%stm)>
<!ELEMENT Assign - - (%id,%exp)>
<!ELEMENT Print - - (%exp*)>

<!ENTITY % exp  "(Id|Num|Op)">
<!ELEMENT Id - - (%id)>
<!ELEMENT Num - - (%int)>
<!ELEMENT Op - - (%exp,%binop,%exp)>
```

A.5 OMG IDL Object Encoding

```
-- http://www.omg.org/corba/corbiiop.htm
enum binop { Plus, Minus, Times, Div};
interface stm {
        enum stm_tag { Compound_tag, Assign_tag, Print_tag};
        attribute stm_tag tag;
}
interface Compound : stm {
        attribute stm head; attribute stm next;
}
interface Assign : stm {
        attribute id string; attribute exp exp;
}
interface Print : stm {
        attribute sequence<exp> args;
}
interface exp {
        enum exp_tag { Id_tag, Num_tag, Op_tag};
        attribute exp_tag tag;
}
interface Id  : exp { attribute id string; }
interface Num : exp { attribute int v; }
interface Op  : exp {
        attribute exp lval; attribute binop bop; attribute exp rval;
}
```