The Zephyr Abstract Syntax Description Language

Daniel C. Wang

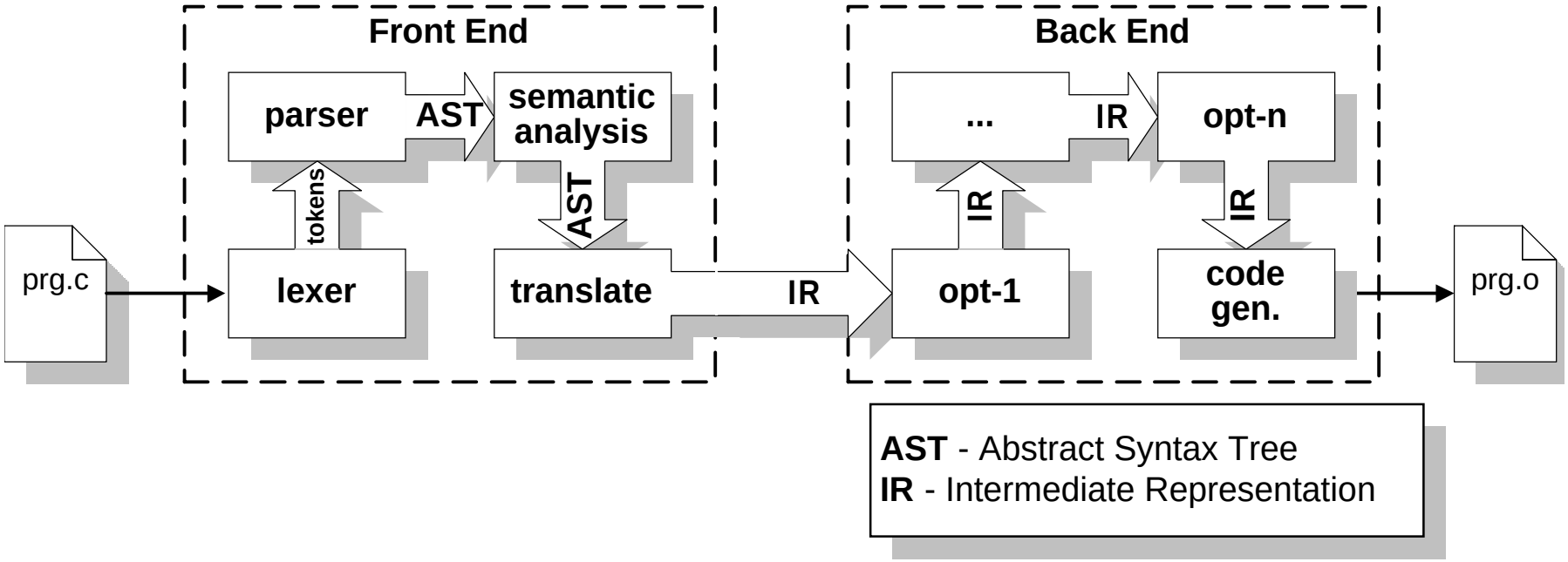
danwang@cs.princeton.edu

Princeton University

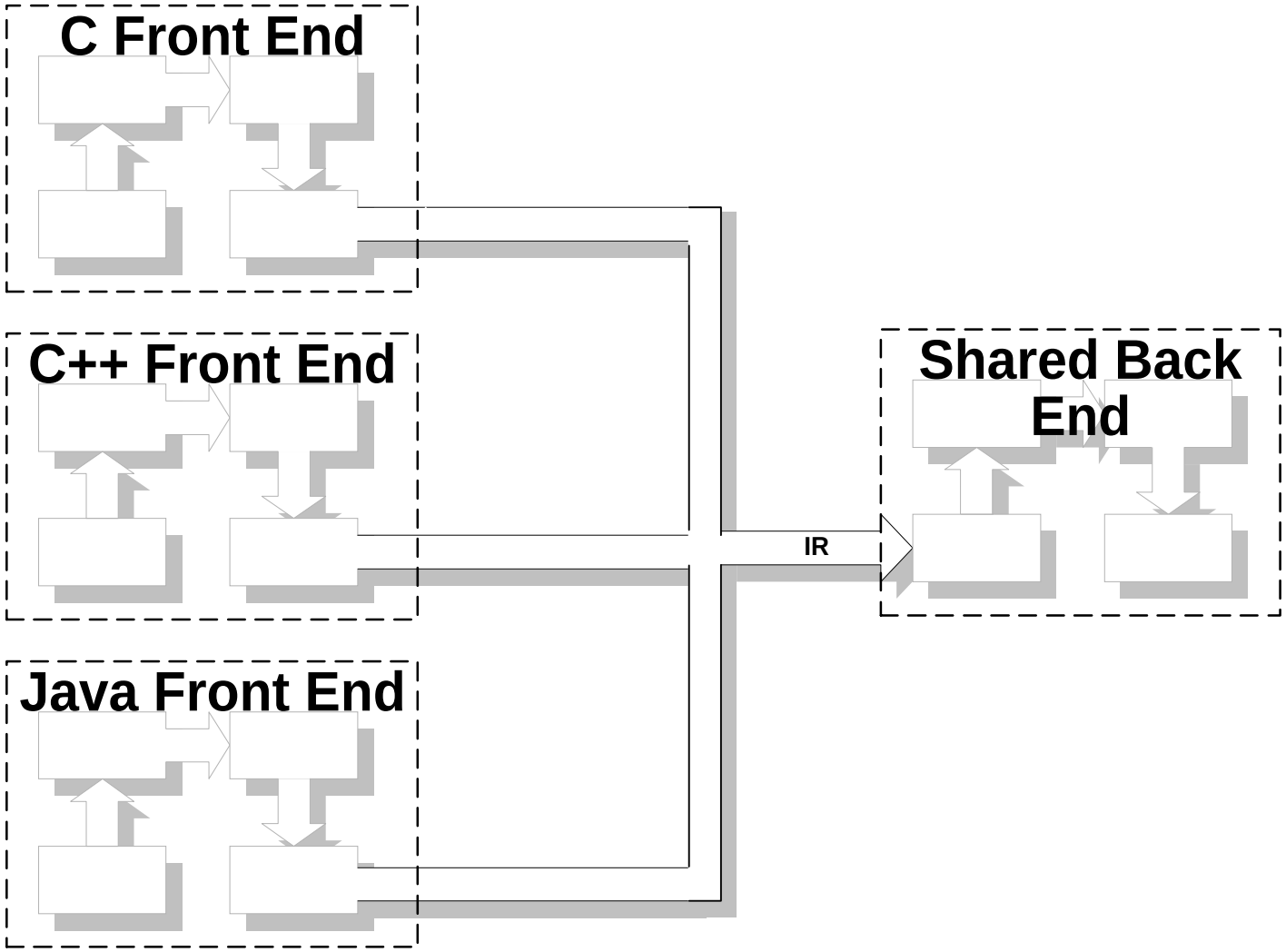
with

Andrew Appel, Jeff Korn, and Chris Serra

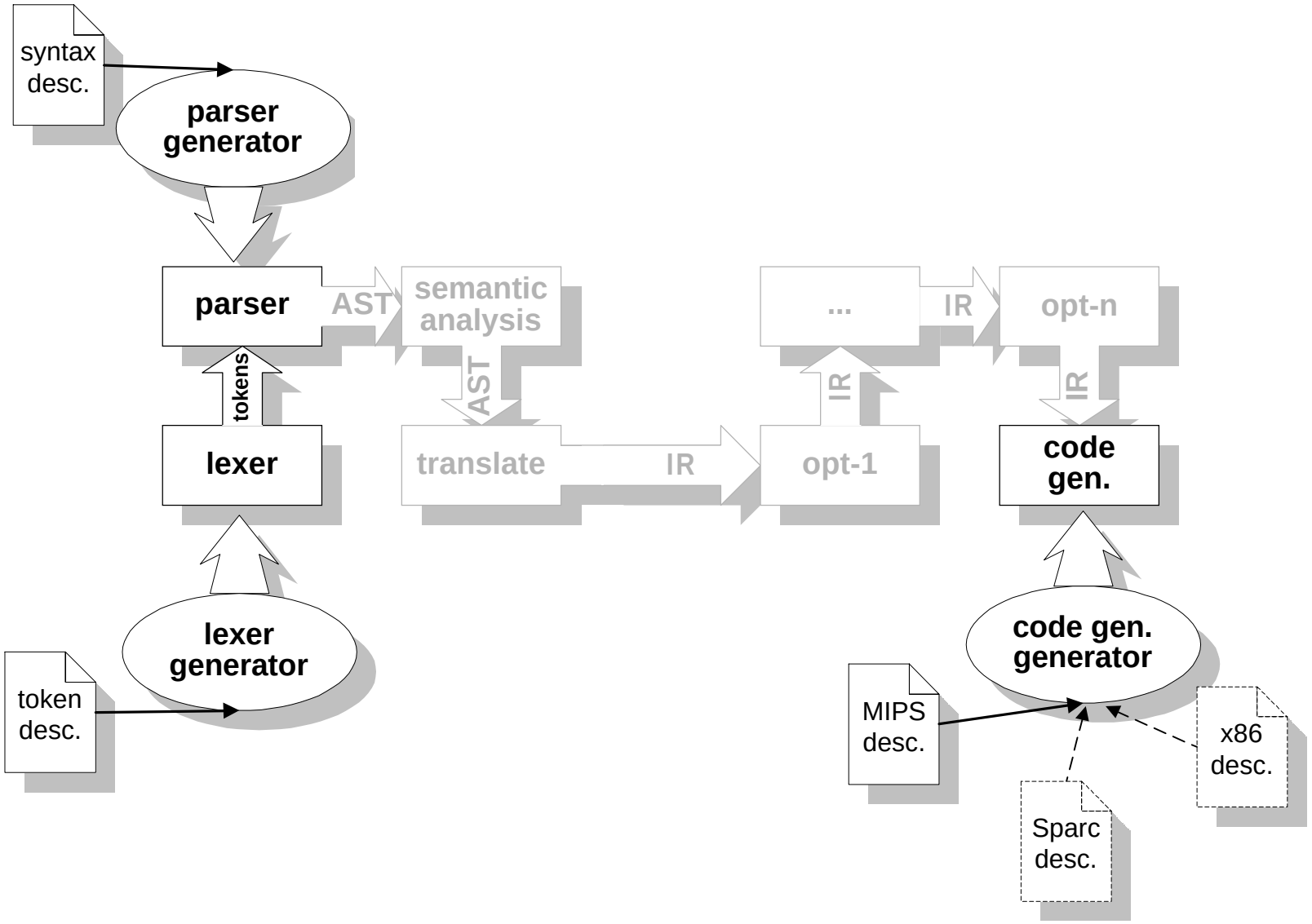
Modular Compilers



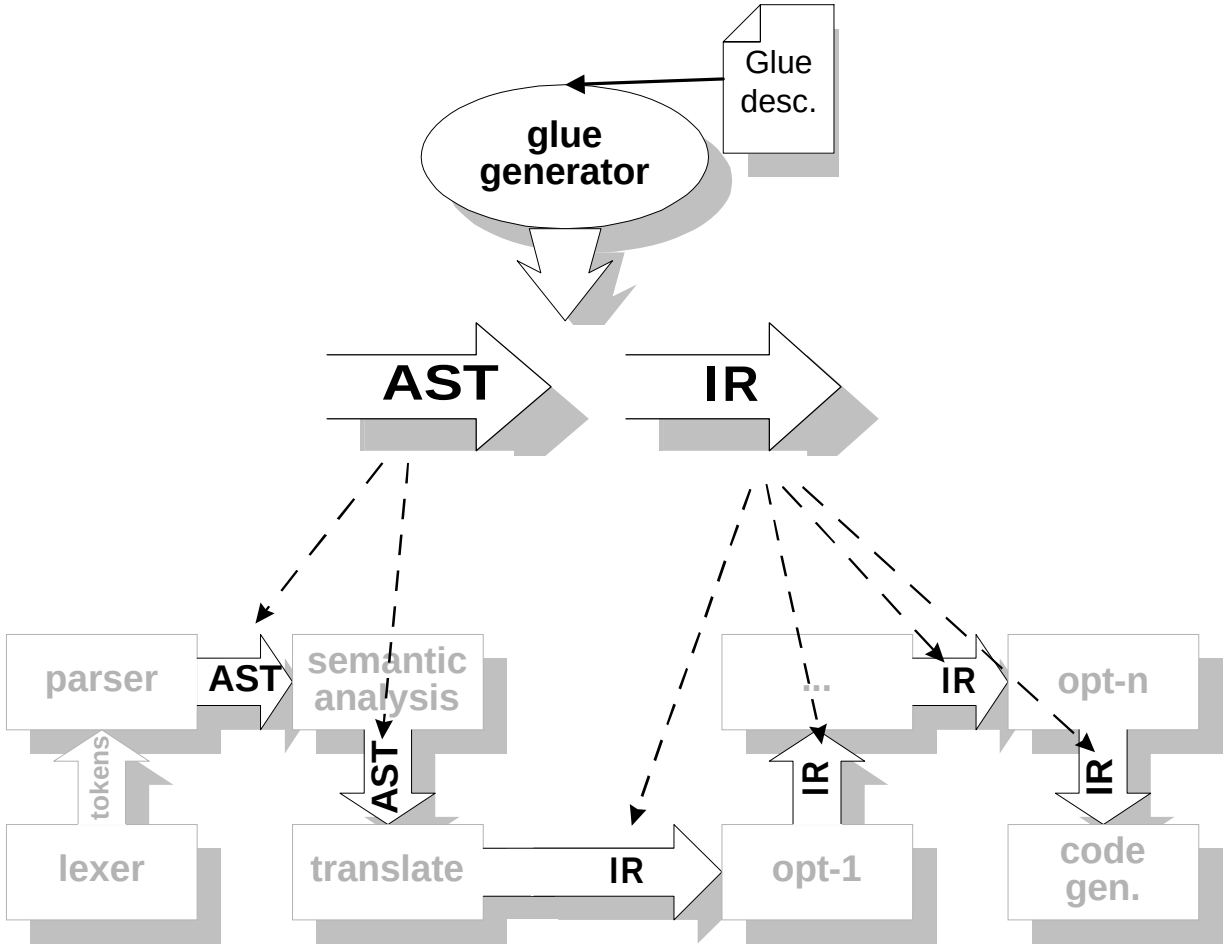
Multi-Language Compiler



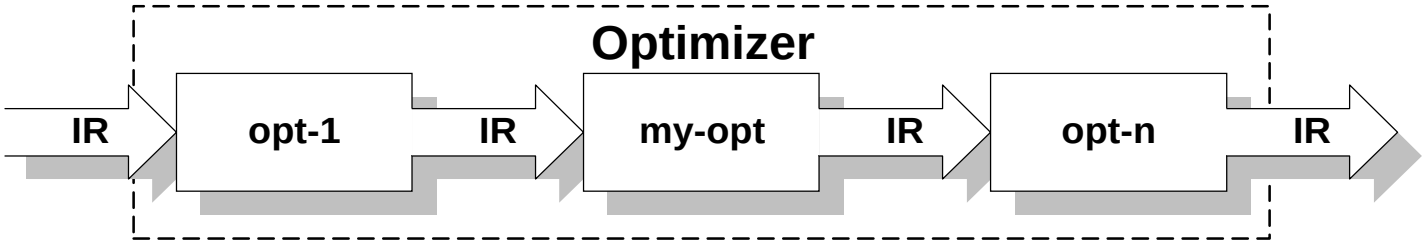
Compiler Construction Tools



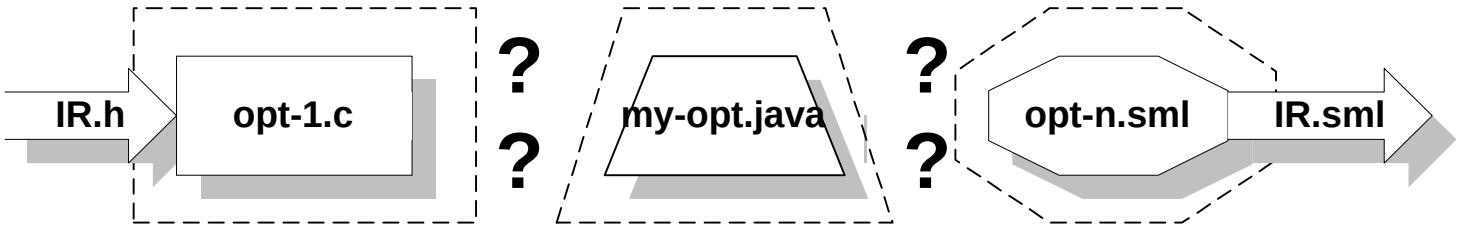
Compiler Glue Construction Tools



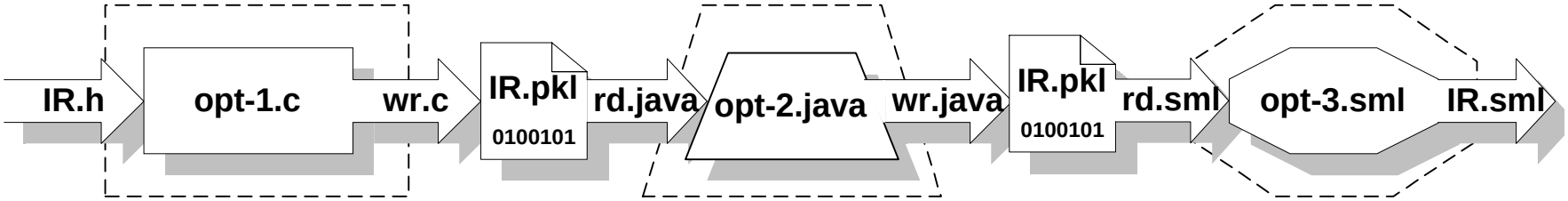
Experimenting with Optimizers



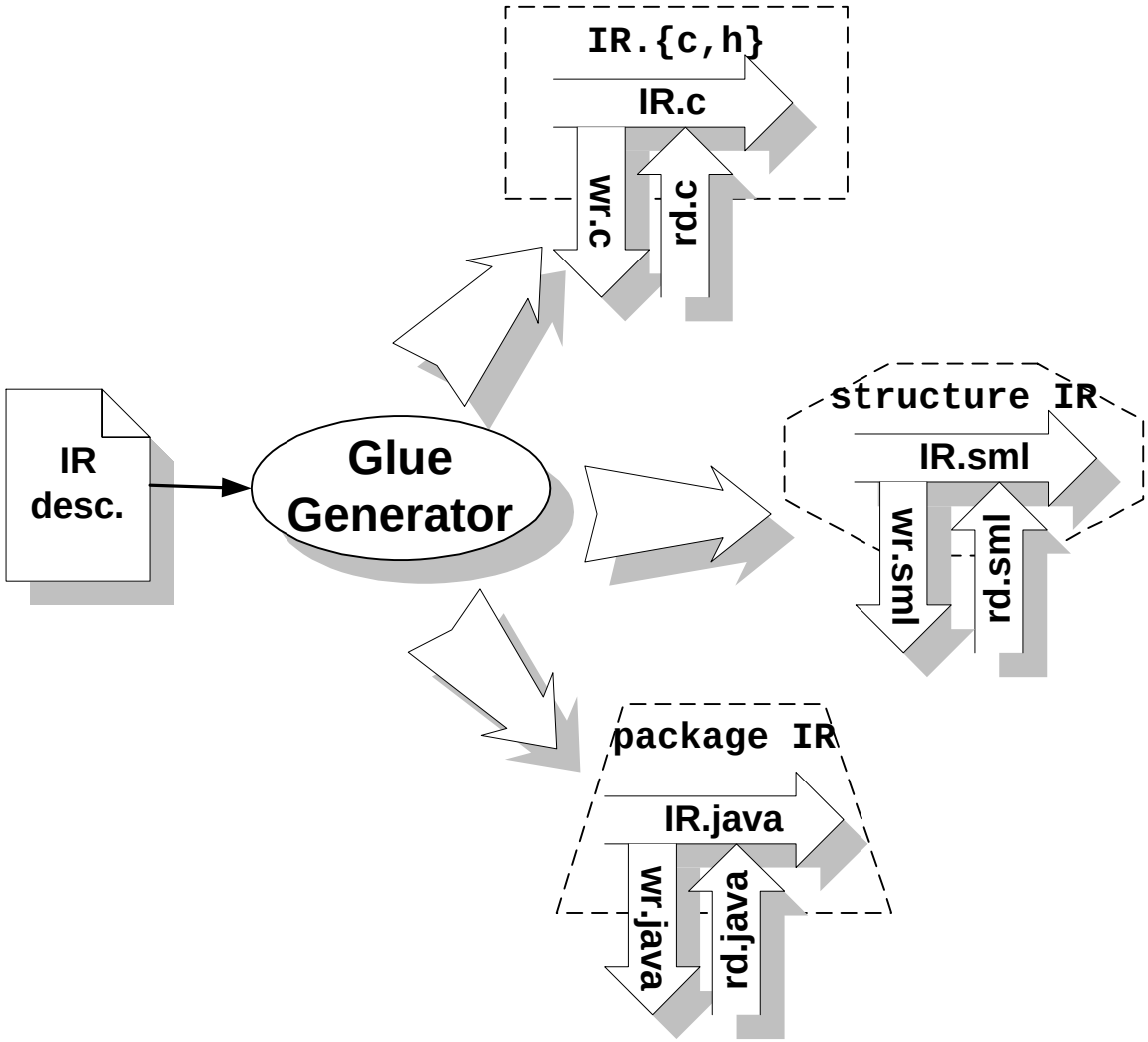
The Optimizer of Babel



Write to "Pickle" Files



Universal Glue Made to Order



Abstract Syntax Description Language

- Serves the same purpose as CORBA's IDL, SGML's DTD or ASN.1
- Describes a class of tree data structures with grammar-like description

Description

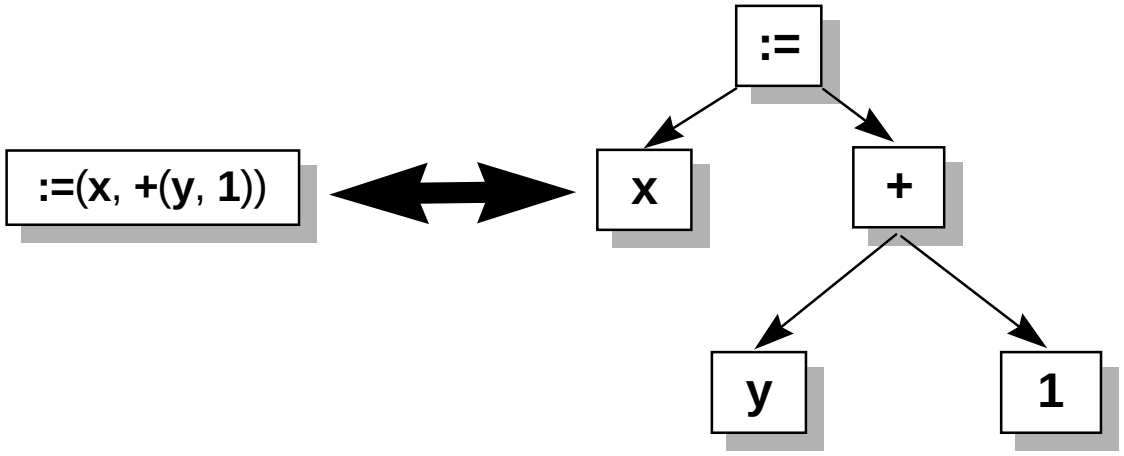
exp = **:=**(*var, exp)*

exp = **+**(*exp, exp)*

exp = *var*

exp = *int*

An Example *exp*



Why Not Use Existing Alternatives?

- CORBA's IDL biased to C types
  - Does not describe recursive structures well.
- SGML too complex for machines
  - Documents described by DTDs can be difficult to parse. HTML hard to describe with LR(1) grammar.
- ASN.1 is a complex language
  - ASN.1 grammar has 300 productions, 150 non-terminals, and semantics defined with 70+ pages.

Design Goals

- Simple
  - More concise and simpler than alternatives
- Generate natural and readable code
  - Programmers must be able to use and read the code
- Produce code for C, C++, Java, and ML
- Deal with existing IR's such as FLINT, SUIF, and lcc's IR

ASDL By Example

Straight Line Program

```
x := 1;
y := x + y;
print(x,y);
```

ASDL Description of SLP ASTs

```
stm = Compound(stm, stm)
    | Assign(identifier, exp)
    | Print(exp_list)

exp_list = ExpList(exp, exp_list)
         | Nil

exp = Id(identifier)
    | Num(int)
    | Op(exp, binop, exp)

binop = Plus | Minus | Times | Div
```

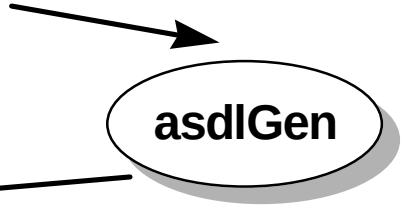
Type

Constructor

Primitives

Generated Code (C like Languages)

```
stm = Compound(stm,stm)
    | Assign(identifier,exp)
    | Print(exp_list)
```



Tagged unions and structs

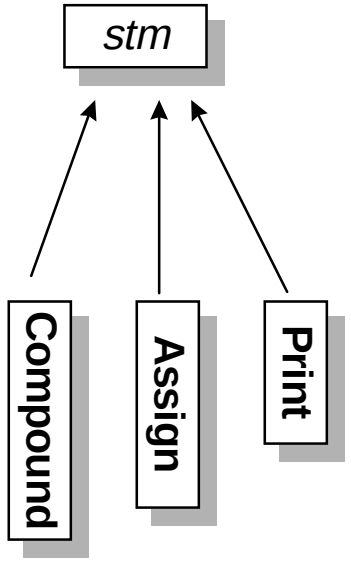
```
typedef struct _stm *stm_ty;
struct _stm {
   enum {Compound_kind=1, Assign_kind=2, Print_kind=3}
   kind;
     union {
      struct { stm_ty stm1; stm_ty stm2; } Compound;
      struct { identifier_ty identifier1;
                        exp_ty exp1; } Assign;
      struct { exp_list_ty exp_list1; } Print;
     } v;
};
stm_ty Compound(stm_ty stm1, stm_ty stm2);
stm_ty Assign(identifier_ty identifier1, exp_ty exp1);
stm_ty Print(exp_list_ty exp_list1);
```

Generated Code (OO Languages)

- Use one level of inheritance

```
abstract public class stm {
   protected int _k;
   public final int kind(){ return _k; }
   public static final int Compound_kind = 1;
   public static final int Assign_kind = 2;
   public static final int Print_kind = 3;
}

public final class Compound extends stm {
    public stm stm1; public stm stm2;
   public Compound(stm stm1, stm stm2){...}
}
public final class Assign extends stm {...}
public final class Print extends stm {...}
```



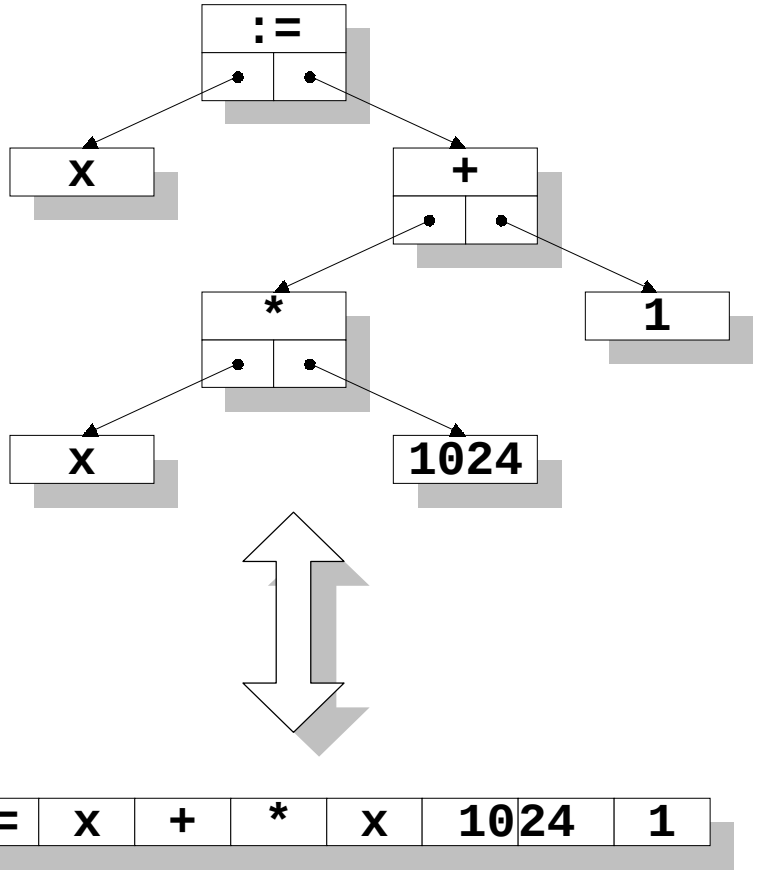
Generated Code (ML)

- Use ML data types

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

Pickle Format

- Binary format
- Simple prefix encoding
- Arbitrary precision integers
- Designed to be concise

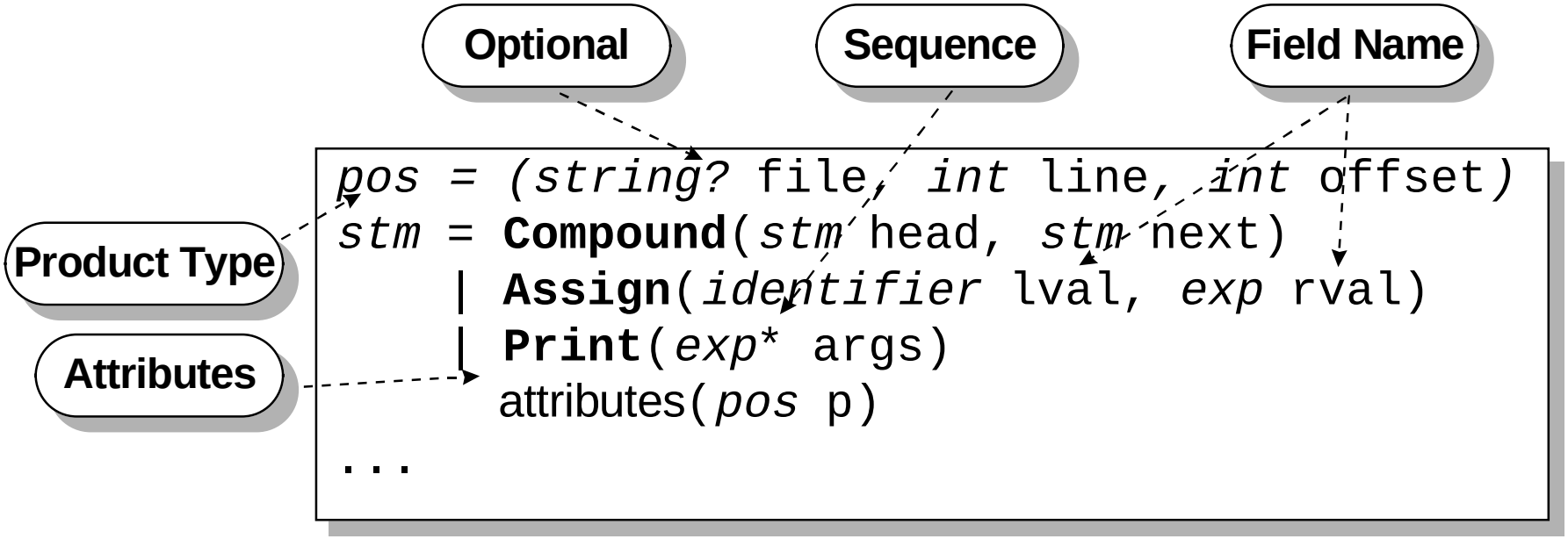


Generated Picklers

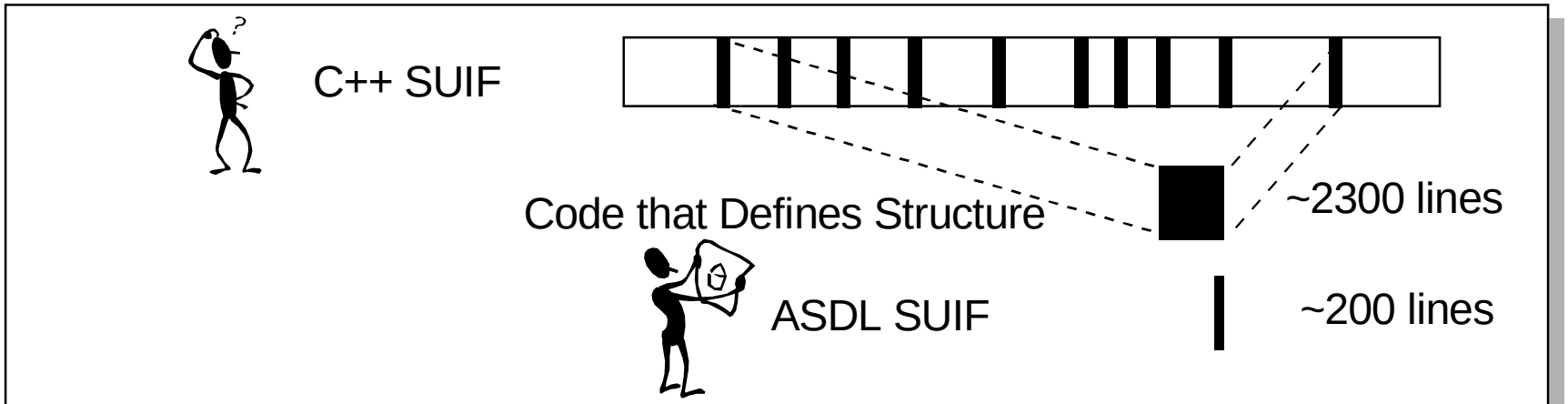
```
...
void pkl_write_stm(stm_ty x, outstream_ty s) {
  switch(x->kind) {
    case Compound_kind:
      pkl_write_int(1, s);
      pkl_write_stm(x->v.Compound.stm1, s);
      pkl_write_stm(x->v.Compound.stm2, s);
      break;
    case Assign_kind:
      pkl_write_int(2, s); ... break;
    case Print_kind: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

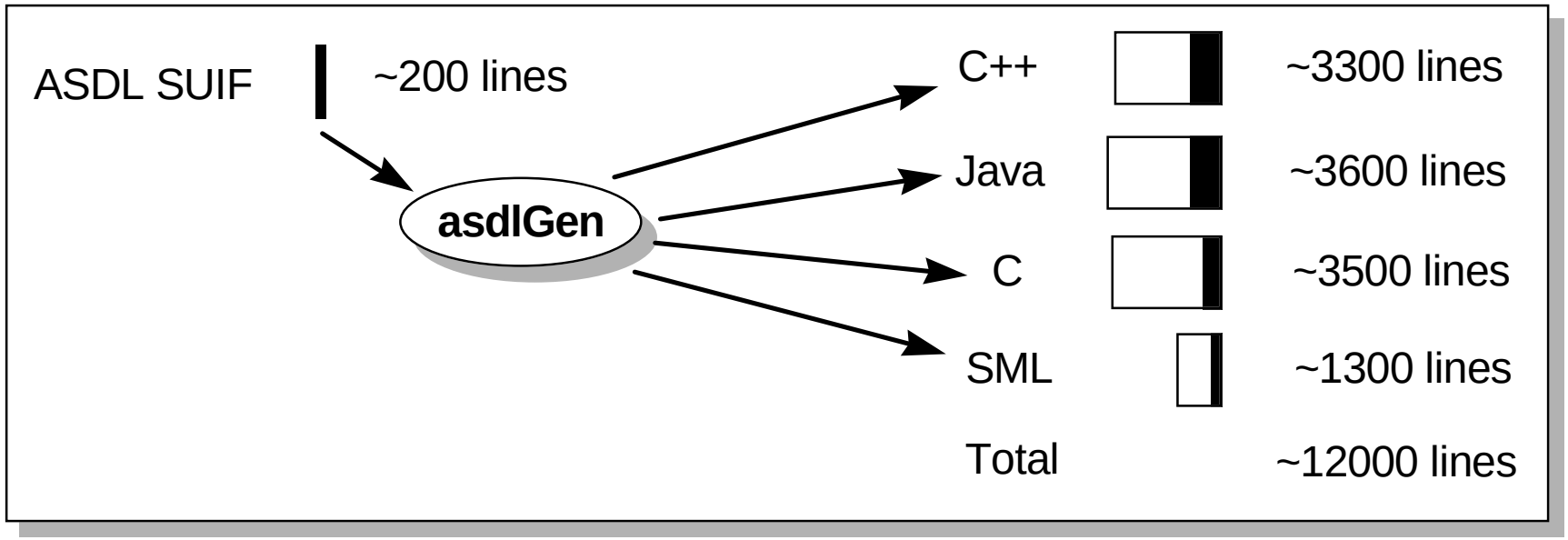
Other Features

- Field names, sequences, and optional
- Product type and attributes



C++ SUIF vs. ASDL SUIF





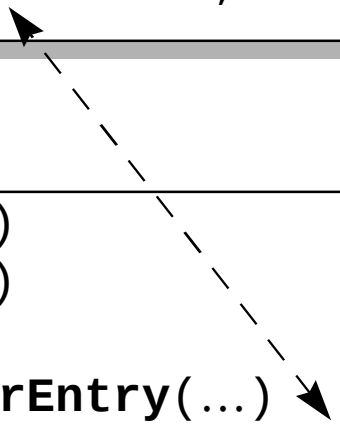
ASDL SUIF Description Not Perfect

- Not all subtyping constraints captured
  - 2 cases
- Awkward encoding of two levels of inheritance
  - 1 case

Problems with Subtyping

```
abstract public class block {…}
public class Basic extends block { …}
public class Funct extends block { …}

abstract public class table_entry {…}
public class VarEntry extends table_entry { … }
public class FunctEntry extends table_entry {
    public Funct value; … }
```



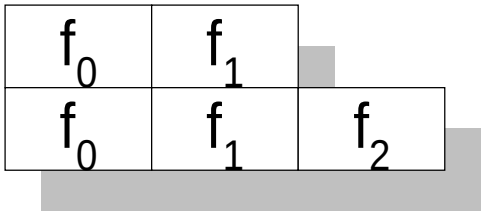
```
block = Basic(…)
      | Funct(…)

table_entry = VarEntry(…)
            | FunctEntry(block,…)
```

More Liberal in ASDL

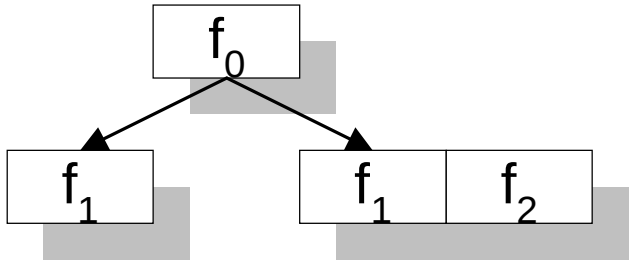
Encoding Inheritance

```
abstract public  class A { ty0 f0; }
public class B extends A { ty1 f1; }
public class C extends B { ty2 f2; }
```



Not Natural in ASDL

```
    a = A(ty0 f0,kids)
kids = B(ty1 f1) | C(ty1 f1, ty2 f2)
```



Modula-3 vs. ASDL Picklers

Modula-3

- Arbitrary graphs
- Modula-3 specific
- Can pickle all Modula-3 types
- Uses runtime type information

ASDL

- Just trees
- Less language specific
- Can pickle only types defined in ASDL
- Uses static type information

Syntax Comparison

ASDL

```
stm = Compound(stm head, stm next)
    | Assign(stm head, stm next)
    | Print(exp* args)
```

CORBA's IDL

```
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;
}
```

Syntax Comparison cont.

SGML DTD

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

ANS.1

```
Stm ::= CHOICE {
    compound SEQUENCE {head Stm, next Stm},
    assign SEQUENCE {head Stm, next Stm},
    print SEQUENCE {args SEQUENCE OF Exp}
}
```

ASDL Status

- **`asdlGen`** produces C, C++, Java, and SML data structures and picklers from ASDL descriptions
- Working on second generation browser
- Full public release with SML sources, binary distribution, and documentation in January
- Web demo and other information at

  htttp://www.cs.princeton.edu/zephyr/ASDL

Future Work

- Modular descriptions
  - Deal with name space management and separate compilation issues
- Separate Description from Implementation
  - Use appropriate language specific implementations rather than concrete descriptions
- Encoding for FLINT, lcc's IR, and improved SUIF 2.0 encoding
- Support for Graphs (??)

National Compiler Infrastructure

Stanford University
Monica Lam et al.
SUIF parallelizing compiler

http://suif.stanford.edu/

Princeton University
Andrew Appel
“Meta-thinking“ and Glue

University of Virgina
Jack Davidson
Norman Ramsey
Very Portable Optimizer (VPO)
Back end generators (CSDL)

http://www.cs.virginia.edu/zephyr/

Description vs. Implementation

Description

```
real = (int mantissa, int exp)
```

Concrete Encoding

```
typedef struct _real* real_ty;
struct _real {
  int mantissa;
  int exp;
}
void pkl_write_real(...) {
   pkl_write_int(x->mantissa, s);
   pkl_write_int(x->exp, s);
}
```

Abstract Encoding

```
typedef double real_ty;
void pkl_write_real(...){
          ??
}
```