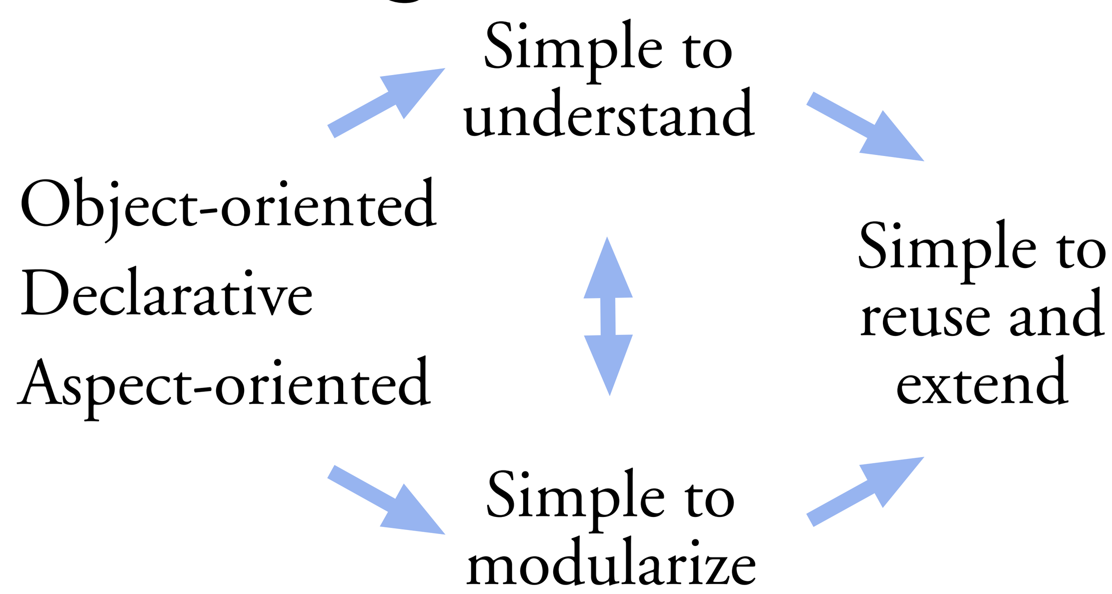
Reuse your compiler!

What if you could reuse your compiler and build your own adapted tools? For example, add coding conventions, add framework support, add domain-specific optimizations, or even add new language constructs... It is possible, with JastAdd II!

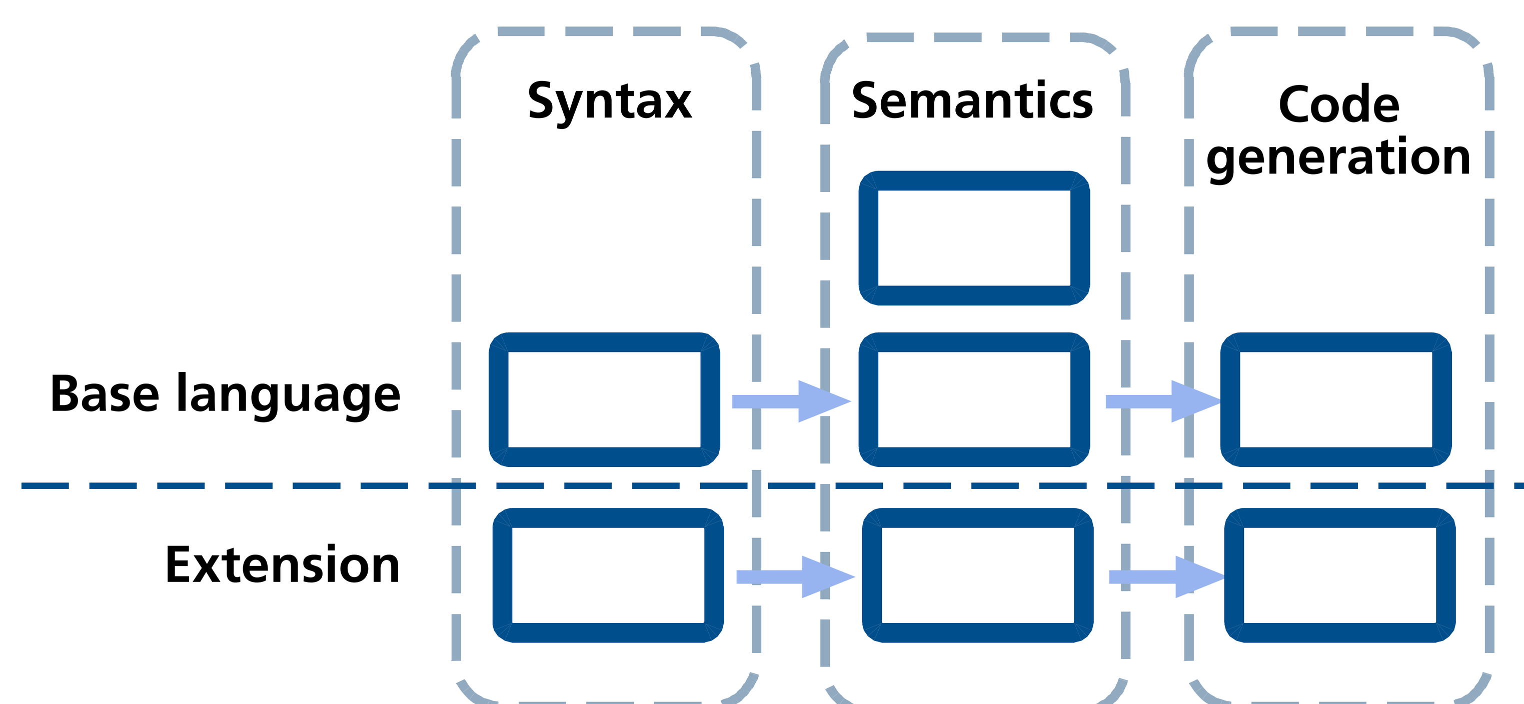
Current status

- complete reusable Java 1.4 compiler
  - bytecode and C backends
  - experimental WCET analysis added
  - extensions to Java 1.5 (generics etc.)
- experimental Rapid compiler
  - force control extension
- many small exerimental compilers (used in compiler construction course)

Advantages



Reuse Architecture



Example scenarios

Coding conventions
Framework conventions

The base language semantics is extended with additional compile-time checks. For example, to support coding conventions such as never assign a local variable more than once, unless it is a sum or a flag used in a loop. A framework-specific convention could be that a particular method must be overridden when specializing a given class

Metrics

To analyze programs for given source code metrics, the base language front end can be reused and extended with specific metrics. For example, compute an upper bound on the worst case execution time (WCET), compute the maximum class hierarchy depth, etc.

Optimizations

Special-purpose optimizations can be added by extending the base language code generation. For example, synthezise dedicated hardware from parts of the software specification.

Simple domain extension

Simple language extensions can be implemented by extending the base language with new syntax and semantics. The new constructs can be mapped back to the base language to reuse the code generation phase. For example, add special-purpose syntax for certain kinds of classes (corresponding to stereotyping in UML).

Full language extension

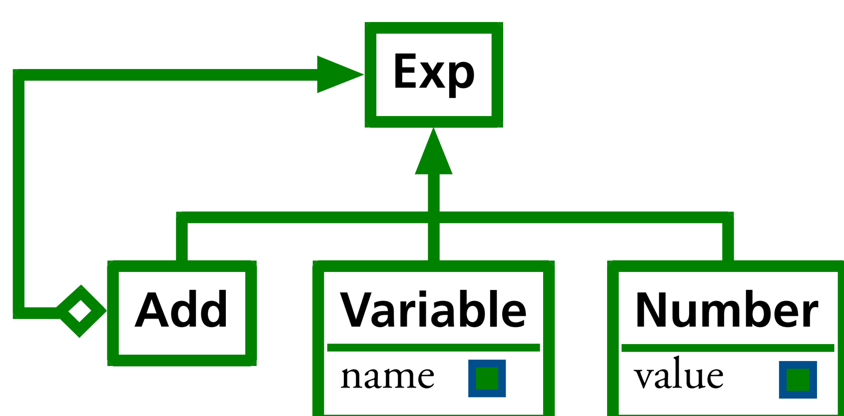
Extend a language with syntax and semantics that cannot be mapped down to the base language, but where code generation needs to be extended as well. For example, adding real-time primitives requires code generation for a real-time run-time system.

Rewritable reference attributed grammars (ReRAGs) – the underlying technology in JastAdd II

The JastAdd II tool weaves the aspect behavior into the class hierarchy, and translates equations and rewrites to Java code that automatically performs attribute evaluation and AST rewriting. All evaluation is transparent: when accessing an AST from outside it will be fully evaluated and rewritten.

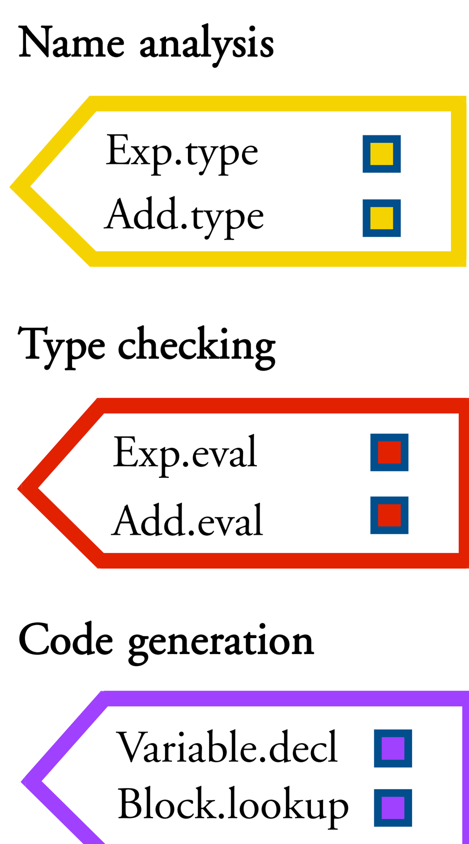
Object-oriented grammar

Superclasses and subclasses correspond to nonterminals and productions. The class hierarchy allows common behavior to be described in superclasses and specialized behavior in subclasses.



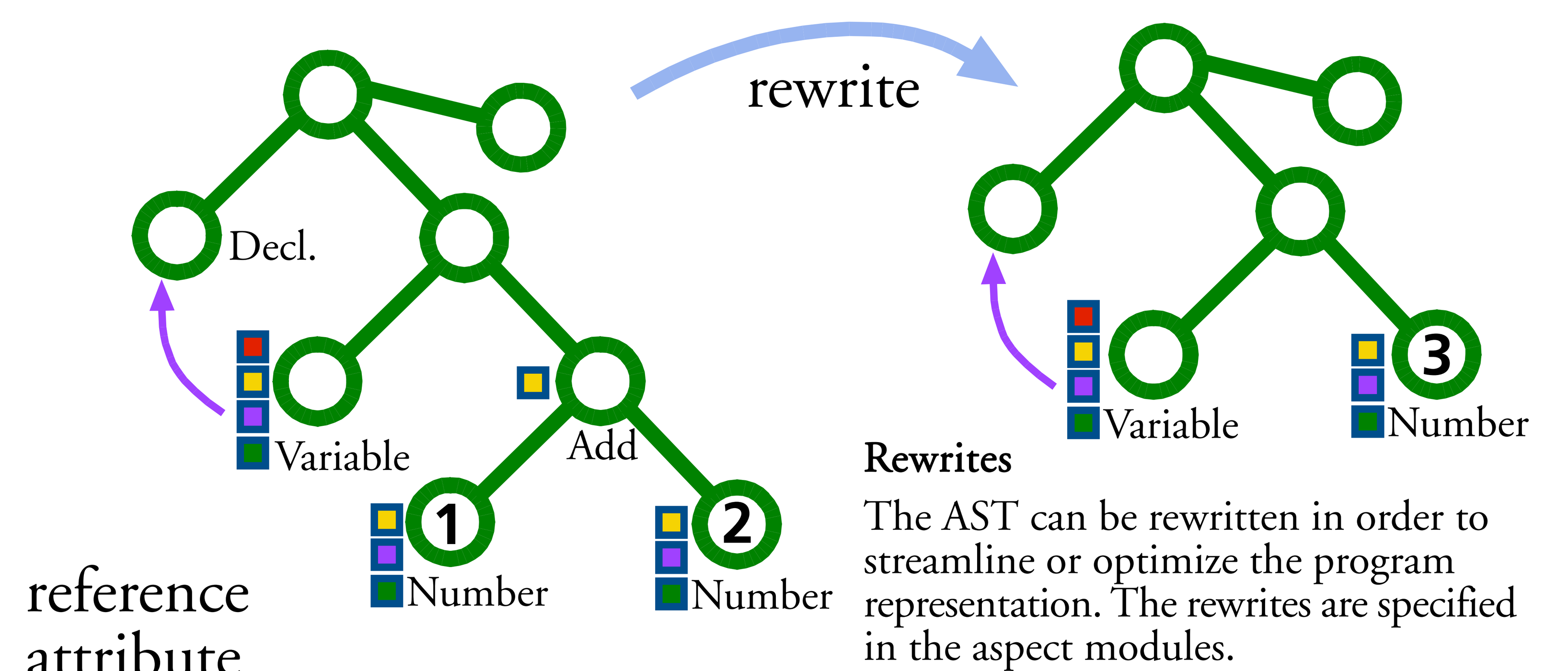
Aspect-oriented modules

A given compiler aspect is often scattered across the class hierarchy. The behavior is modularized by placing it in an aspect module. JastAdd II weaves the aspect module behavior into the class hierarchy.



Attributed abstract syntax tree

The program is represented as an attributed Abstract Syntax Tree (AST). The attribute values are described declaratively by equations in the aspect modules.



Rewrites

The AST can be rewritten in order to streamline or optimize the program representation. The rewrites are specified in the aspect modules.

Reference attribute

Attributes can be references to other AST nodes. This allows, e.g, cross-references from variable use sites to their declarations, to be represented in a straight-forward way.

LUND INSTITUTE OF TECHNOLOGY
Lund University

http://www.cs.lth.se/Research/ProgEnv/rags/

LUCAS
Center for Applied Software Research