The abc scanner and parser, including an LALR(1) grammar for AspectJ

Laurie Hendren, Oege De Moor, Aske Simon Christensen and the abc team

September 13, 2004

1 Introduction

The purpose of this document is to give a clear explanation of the scanner and parser for abc. In defining the scanner and lexer our goal was to come up with clear rules for tokens and to express the grammar in an LALR(1) specification that results in no shift-reduce or reduce-reduce conflicts. Any language extensions implemented using abc should follow the same principles.

We have based our implementation on Polyglot [1], starting with the base Java compiler. The scanner is an extended version of polyglot's base Java scanner. The main difference is that the abc scanner uses states to distinguish between different contexts. The parser uses polyglot's mechanism for extending an existing grammar.

The actual code specifying the scanner and parser can be found in the directory `abc/src/abc/-aspectj/parse` in the files `aspectj.flex` and `aspectj.ppg`.

2 Lexical Structure

The lexical analysis of AspectJ is complicated by the fact that there are really three different languages being parsed: (1) normal Java code, (2)aspect declarations, and (3) pointcut definitions. Each of these three sub-languages has its own lexical structure, and it simplifies the subsequent design of the grammar if the scanner has different states and rules for each sub-grammar.

2.1 Nested Lexical Scopes

From a conceptual point of view we can think of an AspectJ program consisiting of nested lexical scopes. There are four kinds of lexical scopes which we refer to by the mode names, Java, Aspect, Pointcut and PointcutIfExpr. Figure 1 shows an example of all four kinds of scopes and each are discussed in more detail in the subsequent subsections.

2.1.1 Java mode

The outermost scope is always has Java mode. In this mode all tokens are scanned exactly as in Java, with the exception that **privileged**, **aspect** and **pointcut** are considered to be keywords and cannot be used an identifiers.

```
import java.lang.*;

class OrdinaryJavaClass {
   public int x;
   public int y;

   String foo(int x)
       { return ("The value of x + 1 is " + (x + 1));
       }
}

privileged aspect /* a privileged aspect with a per declaration in the header */

    OrdinaryAspect percflow ( call(void Foo.m()) )
    {
      /* declare declaration */
      declare warning: call(*1.Foo+.new(..)): "pkg ending in 1, class or subclass of Foo";

      /* pointcut declarations */
      pointcut notKeywords(): call(void *if*..*while*(int,boolean,*for*));

      pointcut hasSpecialIf(): if (Tracing.isEnabled());

      /* advice declaration */
      after (Point p) returning(int x): target(p) && call(int getX())
        {
          System.out.println( "Returning int value " + x + " for p = " + p);
        }

      /* inter-type member declaration */
      int OrdinaryJavaClass.incr2(int i)
          { return(x+2);
          }

      /* ordinary Java declarations */
      int x;

      static int incr3(int x)
        { return(x+3);
        }

      /* a nested class */
      class

          NestedClass {
          /* In Java mode, after and before are not keywords. */
            public int after;
            public int getBefore()
            { return(OtherClass.before);
            }
        } // end of NestedClass

    } // end of OrdinaryAspect
```


Figure 1: AspectJ code with nested lexical scopes

2.1.2 Aspect mode

Inside the program one can have a nested Aspect scope which begins just after the keyword **aspect** and ends at the end of the aspect's body. Figure 1 shows one Aspect scope corresponding to the body of the declaration of the aspect named "OrdinaryAspect".

In Aspect mode all symbols are exactly the same as in Java mode, except for the addition of keywords **after**, **around**, **before**, **declare**, **issingleton**, **percflow**, **percflowbelow**, **pertarget**, **perthis**, **pointcut**, **proceed**. Just like normal Java keywords, these additional keywords cannot be used as identifiers, inside of an Aspect scope.

2.1.3 Pointcut mode

Whereas the Java and Aspect modes are very similar, basically differing only on the keywords recognized, the Pointcut mode has a completely different lexical structure. In Figure 1, the lexical scopes for pointcuts are shown by the boxes nested inside the Aspect. There are four contexts in which pointcut scopes occur, as follows:

Pointcut Context #1 - a Per clause in an aspect declaration: The header of an aspect declaration ends with an an optional per clause. A per clause consists of either the keyword **issingleton** or a parenthesized pointcut expression preceded by one of the keywords **percflow**, **percflowbelow**, **pertarget** or **perthis**. A Pointcut scope starts after one of the per keywords and ends at the matching closing parenthesis that surrounds the pointcut. Figure 1 shows a nested scope for a pointcut expression following the **percflow** keyword.

Pointcut Context #2 - body of a declare declaration: Inside the body of an aspect one can define a declare declaration. A lexical Pointcut scope begins just after the keyword **declare** and ends at the **;** terminating the declaration.

The example program shows a declaration of a **warning** which matches all calls to constructors of classes found in packages ending in the digit 1, with classname Foo or a subclass of Foo.

Pointcut Context #3 - body of a pointcut declaration: Pointcut declarations are provided in AspectJ as a way of defining a named pointcut. In the example program in Figure 1 two such declarations are given, one for *notKeywords* and another for *hasSpecialIf*. Inside pointcut declarations a pointcut lexical scope begins immediately following the **pointcut** keyword and ends after the **;** terminating the pointcut declaration. Pointcut declarations can appear both inside aspects and inside ordinary Java classes.

Pointcut Context #4 - header of an advice declaration: Advice declarations have a pointcut expression in their header. All such pointcuts will be preceded by one of the keywords **before**, **after** or **around**. For example, in Figure 1, a pointcut follows an **after** keyword. The pointcut ends before the body of the pointcut begins, signalled by a **{**.

Thus, a pointcut context starts immediately after a **before**, **after** or **around** token, and ends at the first opening brace encountered.

2.1.4 PointcutIfExpr mode

Pointcuts have no lexical scopes nested inside them, except for one case, the **if** pointcut. The **if** pointcut contains a boolean expression, which is just Java code. We define a special mode called PointcutIfExpr which starts right after the **if** keyword inside a pointcut, and ends at the terminating parenthesis closing the boolean expression. The lexical structure, in terms of tokens recognized, is identical to the Aspect mode. However, in the implementation of the scanner, the end of PointcutIfExpr mode always signals a return to Pointcut mode.

In Figure 1, the pointcut declaration named *hasSpecialIf* shows an example of a nested PointcutIfExpr lexical scope.

2.2 Nested Aspects, Classes and Interfaces

In AspectJ classes, interfaces and aspects may be nested inside each other. In terms of nested lexical scope, a new scope is entered each time the keywords **class**, **interface** and **aspect** are entered, and the scope is exited at the closing right brace. In the case of **class** and **interface** the scope entered has Java mode, whereas for the case of **aspect** the scope entered has Aspect mode.

At the bottom of Figure 1 we give the declaration of a nested class called *NestedClass*. Note that inside the class declaration is in Java mode, so the keywords recognized are those corresponding to Java mode (i.e. Java keywords plus **aspect**, **privileged** and **pointcut**). Thus, in the example, *before* and *after* are considered identifiers, not keywords. Note that this use of inner classes provides a mechanism for referring to variables defined in other classes that may have the same name as keywords in Aspect mode. In our example we have defined the method *getBefore* to read the value of *OtherClass.before*.

The above rule for entering a new lexical scope upon encounter of the keyword **class** is complicated by the fact that **class** does not always signal a new class declaration in Java. In particular, it can be used to return a *Class* object that represents a type, as in *C.class* (this is useful, for example, to create typed lists, where the intended element type is stored with the list structure). All such uses of the **class** keyword are preceded by a dot, and class declarations themselves are never preceded by a dot. For that reason, the lexer records whether the last emitted token was a dot; if it is, then the **class** keyword does *not* cause a transition to a new lexical scope.

2.3 Lexical Structure of Pointcuts

The language for defining pointcuts is a very special purpose language that provides a way of specifying identifier patterns, classname patterns, and more complex expressions involving patterns.

2.3.1 Examples of differences from the Java lexical structure

The example program clearly demonstrates ways in which the lexical structure of pointcuts is very different from the lexical structure of Java.

For example, if one were to use the ordinary Java lexical rules, then the expression "*1.Foo+.new(..)" would be tokenized as:

`[ op("*"), fp_literal(1.0), Id("Foo"), op("+"), op("."), keyword("new"), op("("), op("."), op(".") op(")")].`

However, in pointcuts, the intended lexical structure is quite different and would be tokenized as:

`[ IdPat("*1"), op("."), id("Foo"), op("+"), op("."), keyword("new"), op("("), op(".."), op(")")].`

Note that "*1" is an identifier pattern, which matches any identifier ending in 1. Also note, the sequence ".." is recognized as one token, which also simplifies the grammar.

Another example of the need for a special lexical structure for pointcuts is given in the definition of the pointcut *notKeywords*. The ordinary Java lexical rules would tokenize the expression "*if*..*while*" as:

`[ op("*"), keyword("if"), op("*"), op("."), op("."), op("*"), keyword("while"), op("*")].`

whereas this expression inside a pointcut has a completely different lexical structure, namely:

`[ IdPat("*if*"), op(".."), IdPat("*while*") ] .`

2.3.2 Tokens in pointcuts

Since the Java lexical structure clearly doesn't match the pointcut language very well, a completely different lexical structure is defined for pointcuts. This can be summarized as follows.

Keywords: All of the keywords in Java mode (including **aspect** and **privileged**), plus the following: **adviceexecution**, **args**, **call**, **cflow**, **cflowbelow**, **error**, **execution**, **get**, **handler**, **initialization**, **parents**, **precedence**, **preinitialization**, **returning**, **set**, **soft**, **staticinitialization**, **target**, **throwing**, **warning**, **within**, **withincode**.

Note that extra keywords in Aspect mode, such as **before**, are not keywords in the Pointcut mode, and similarly the extra keywords in Pointcut mode are not keywords in Aspect mode.

Symbols: The symbols recognized in pointcuts are: `op("(")`, `op(")")`, `op("[")`, `op(".")`, `op(",")`, `op(":")`, `op(";")`, `op("{")`, `op("ï")`, `op("!")`, `op("&&")`, `op("||")`, `op("..")`, `op("+")`.

Identifiers and Identifier Patterns: Identifiers are matched using the same regular expression as in Java mode, namely:

```
Identifier = [:jletter:][:jletterdigit:]*
```

Identifier patterns are recognized as:

```
IdentifierPattern = ( "*" | [:jletter:]  )  ( "*" | [:jletterdigit:]  )*
```

Since identifiers and identifier patterns are used in pointcuts to specify names that may occur anywhere, including Java code that has been defined in a library, it is possible that a pattern might want to refer to something with the same name as one of the extra keywords. This is handled later by the grammar, where the extra keywords are explicitly allowed as one alternative of the rule for *simple_name_pattern*, see Section 3.5.1.

3 LALR(1) Grammar

In this section we outline the grammar of AspectJ. If you have a colour version of this document, you will see that all references to productions in the original Java grammar are given in red. The base Java grammar was originally developed by Scott Ananian and is distributed with Polyglot.

In terms of the polyglot implementation, all red productions are part of the base Java grammar, whereas the blue productions are those that are added as part of `abc`'s AspectJ grammar.

The `abc` AspectJ grammar is LALR(1) with no shift-reduce or reduce-reduce conflicts. In order to achieve this conflict-free grammar there are several places where a slightly too large language is specified, and these are places where further weeding must be used to weed out invalid programs.

3.1 Extensions to the Java Grammar

The following five rules are already found in the Java grammar. The alternatives given below are additional alternatives to those rules. At the highest level (*type_declaration*), we add the possibility for declaring an aspect. Inside a class (*class_member_declaration*) and inside an interface (*interface_member_declaration*), we add the possibility of declaring an aspect or a pointcut. Finally, we add the special method call for proceed.

⟨*type_declaration*⟩ ::= ⟨*aspect_declaration*⟩

⟨*class_member_declaration*⟩ ::= ⟨*aspect_declaration*⟩
 | ⟨*pointcut_declaration*⟩

⟨*interface_member_declaration*⟩ ::= ⟨*aspect_declaration*⟩
 | ⟨*pointcut_declaration*⟩

⟨*method_invocation*⟩ ::= 'proceed' '(' ⟨*argument_list_opt*⟩ ')'

3.2 Aspect Declaration

An aspect declaration has a header where the modifiers may include **privileged**. We keep the **privileged** keyword separate from all other modifiers since it can only be used in this context.

⟨*aspect_declaration*⟩ ::=
 ⟨*modifiers_opt*⟩ 'privileged' ⟨*modifiers_opt*⟩ 'aspect' IDENTIFIER ⟨*super_opt*⟩ ⟨*interfaces_opt*⟩ ⟨*perclause_opt*⟩ ⟨*aspect_body*⟩
 | ⟨*modifiers_opt*⟩ 'aspect' IDENTIFIER ⟨*super_opt*⟩ ⟨*interfaces_opt*⟩ ⟨*perclause_opt*⟩ ⟨*aspect_body*⟩

3.2.1 Per Clause

An aspect declaration has an optional per clause. Note that this is one place in the grammar where pointcut expressions are introduced. The last alternative has been introduced for compatibility with `ajc`.

⟨*perclause_opt*⟩ ::= ϵ | ⟨*perclause*⟩

⟨*perclause*⟩ ::=
'pertarget' '(' ⟨*pointcut_expr*⟩ ')'
| 'perthis' '(' ⟨*pointcut_expr*⟩ ')'
| 'percflow' '(' ⟨*pointcut_expr*⟩ ')'
| 'percflowbelow' '(' ⟨*pointcut_expr*⟩ ')'
| 'issingleton'
| 'issingleton' '(' ')'

3.2.2 Aspect Body

An aspect body consists of zero or more declarations. These include all valid *class_body_declarations*, plus three new kinds of declarations specific to AspectJ. Note that

⟨*aspect_body*⟩ ::= '' '' | '' ⟨*aspect_body_declarations*⟩ ''

⟨*aspect_body_declarations*⟩ ::=
⟨*aspect_body_declarations*⟩
| ⟨*aspect_body_declarations*⟩ ⟨*aspect_body_declaration*⟩

⟨*aspect_body_declaration*⟩ ::=
⟨*class_body_declaration*⟩
| ⟨*declare_declaration*⟩
| ⟨*advice_declaration*⟩
| ⟨*intertype_member_declaration*⟩

3.3 Aspect Body Declarations

3.3.1 Declare Declarations

⟨*declare_declaration*⟩ ::=
'declare' 'parents' ':' ⟨*classname_pattern_expr*⟩ 'extends' ⟨*class_type_list*⟩ ';'
| 'declare' 'parents' ':' ⟨*classname_pattern_expr*⟩ 'implements' ⟨*interface_type_list*⟩
| 'declare' 'warning' ':' ⟨*pointcut_expr*⟩ ':' STRINGLITERAL ';'
| 'declare' 'error' ':' ⟨*pointcut_expr*⟩ ':' STRINGLITERAL ';'
| 'declare' 'soft' ':' ⟨*pointcut_expr*⟩ ';'
| 'declare' 'precedence' ':' ⟨*classname_pattern_expr_list*⟩ ';'

3.3.2 Pointcut Declarations

⟨*pointcut_declaration*⟩ ::=
  ⟨*modifiers_opt*⟩ 'pointcut' IDENTIFIER '(' ⟨*formal_parameter_list_opt*⟩ ')' ';'
| ⟨*modifiers_opt*⟩ 'pointcut' IDENTIFIER '(' ⟨*formal_parameter_list_opt*⟩ ')' ':' ⟨*pointcut_expr*⟩ ';'

Note, a later weeding phase must ensure that:

- For the first alternative the modifiers must include **abstract**.
- For the second alternative the modifiers must not include **abstract**.

3.3.3 Advice Declarations

⟨*advice_declaration*⟩ ::=
  ⟨*modifiers_opt*⟩ ⟨*advice_spec*⟩ ⟨*throws_opt*⟩ ':' ⟨*pointcut_expr*⟩ ⟨*method_body*⟩

⟨*advice_spec*⟩ :=
  'before' '(' ⟨*formal_parameter_list_opt*⟩ ')'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')' 'returning'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')' 'returning' '(' ')'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')' 'returning' '(' ⟨*formal_parameter*⟩ ')'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')' 'throwing'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')' 'throwing' '(' ')'
| 'after' '(' ⟨*formal_parameter_list_opt*⟩ ')' 'throwing' '(' ⟨*formal_parameter*⟩ ')'
| ⟨*type*⟩ 'around' '(' ⟨*formal_parameter_list_opt*⟩ ')'
| 'void' 'around' '(' ⟨*formal_parameter_list_opt*⟩ ')'

Notes:

- The only valid modifier for an *advice_declaration is* **strictfp**.
- *The superfluous parentheses in the second alternatives of returning and throwing have been introduced for compatibility with* `ajc`.

3.3.4 Intertype Member Declarations

⟨*intertype_member_declaration*⟩ ::=
  ⟨*modifiers_opt*⟩ 'void' ⟨*name*⟩ '.' IDENTIFIER '(' ⟨*formal_parameter_list_opt*⟩ ')' ⟨*throws_opt*⟩ ⟨*method_body*⟩
| ⟨*modifiers_opt*⟩ ⟨*type*⟩ ⟨*name*⟩ '.' IDENTIFIER '(' ⟨*formal_parameter_list_opt*⟩ ')' ⟨*throws_opt*⟩ ⟨*method_body*⟩
| ⟨*modifiers_opt*⟩ ⟨*name*⟩ '.' 'new' '(' ⟨*formal_parameter_list_opt*⟩ ')' ⟨*throws_opt*⟩ ⟨*constructor_body*⟩
| ⟨*modifiers_opt*⟩ ⟨*type*⟩ ⟨*name*⟩ '.' IDENTIFIER ';'
| ⟨*modifiers_opt*⟩ ⟨*type*⟩ ⟨*name*⟩ '.' IDENTIFIER '=' ⟨*variable_initializer*⟩ ';'

3.4 Pointcut Expressions

⟨*pointcut_expr*⟩ ::=
  ⟨*or_pointcut_expr*⟩
| ⟨*pointcut_expr*⟩ '&&' ⟨*or_pointcut_expr*⟩

⟨*or_pointcut_expr*⟩ ::=
  ⟨*unary_pointcut_expr*⟩
| ⟨*or_pointcut_expr*⟩ '||' ⟨*unary_pointcut_expr*⟩

⟨*unary_pointcut_expr*⟩ ::=
  ⟨*basic_pointcut_expr*⟩
| '!' ⟨*unary_pointcut_expr*⟩

⟨*basic_pointcut_expr*⟩ ::=
  '(' ⟨*pointcut_expr*⟩ ')'
| 'call' '(' ⟨*method_constructor_pattern*⟩ ')'
| 'execution' '(' ⟨*method_constructor_pattern*⟩ ')'
| 'initialization' '(' ⟨*constructor_pattern*⟩ ')'
| 'preinitialization' '(' ⟨*constructor_pattern*⟩ ')'
| 'staticinitialization' '(' ⟨*classname_pattern_expr*⟩ ')'
| 'get' '(' ⟨*field_pattern*⟩ ')'
| 'set' '(' ⟨*field_pattern*⟩ ')'
| 'handler' '(' ⟨*classname_pattern_expr*⟩ ')'
| 'adviceexecution' '(' ')'
| 'within' '(' ⟨*classname_pattern_expr*⟩ ')'
| 'withincode' '(' ⟨*method_constructor_pattern*⟩ ')'
| 'cflow' '(' ⟨*pointcut_expr*⟩ ')'
| 'cflowbelow' '(' ⟨*pointcut_expr*⟩ ')'
| 'if' '(' ⟨*expression*⟩ ')'
| 'this' '(' ⟨*type_id_star*⟩ ')'
| 'target' '(' ⟨*type_id_star*⟩ ')'
| 'args' '(' ⟨*type_id_star_list_opt*⟩ ')'
| ⟨*name*⟩ '(' ⟨*type_id_star_list_opt*⟩ ')'

3.5 Patterns

3.5.1 Name Patterns

In this section we give the rules for specifying names as patterns. The grammar explicitly allows the extra keywords introduced for AspectJ to be a valid *simple_name_pattern*.

⟨*name_pattern*⟩ ::=
  ⟨*simple_name_pattern*⟩
| ⟨*name_pattern*⟩ '.' ⟨*simple_name_pattern*⟩
| ⟨*name_pattern*⟩ '..' ⟨*simple_name_pattern*⟩

⟨*simple_name_pattern*⟩ ::=
  '*'
| IDENTIFIER
| IDENTIFIERPATTERN
| ⟨*aspectj_reserved_identifier*⟩

⟨*aspectj_reserved_identifier*⟩ ::=
  'aspect' | 'privileged'
| 'adviceexecution' | 'args' | 'call' | 'cflow' | 'cflowbelow' | 'error'
| 'execution' | 'get' | 'handler' | 'initialization' | 'parents'
| 'precedence' | 'preinitialization' | 'returning' | 'set'
| 'soft' | 'staticinitialization' | 'target' | 'throwing'
| 'warning' | 'withincode'

We also require two special name patterns to distinguish between the cases when the pattern terminates with an identifier or the token `new`. Note that in the next two grammar rules we allow a parenthesized *type_pattern_expression* when we really want to allow only a *classname_pattern_expression*. This is required to make the grammar for *method_pattern* and *constructor_pattern* LALR(1), and must be checked at weeding time.

⟨*classtype_dot_id*⟩ ::=
  ⟨*simple_name_pattern*⟩
| ⟨*name_pattern*⟩ '.' ⟨*simple_name_pattern*⟩
| ⟨*name_pattern*⟩ '+' '.' ⟨*simple_name_pattern*⟩
| ⟨*name_pattern*⟩ '..' ⟨*simple_name_pattern*⟩
| '(' ⟨*type_pattern_expr*⟩ ')' '.' ⟨*simple_name_pattern*⟩

⟨*classtype_dot_new*⟩ ::=

| 'new'
| ⟨*name_pattern*⟩ '.' 'new'
| ⟨*name_pattern*⟩ '+' '.' 'new'
| ⟨*name_pattern*⟩ '..' 'new'
| '(' ⟨*type_pattern_expr*⟩ ')' '.' 'new'

3.5.2 Type Pattern Expressions

This section defines type pattern expressions. These are a superset of class name pattern expressions. The main difference is what is allowed at the leaves of the pattern. In the case of class name patterns the leaves were name patterns, whereas with type pattern expressions the leaves can be any valid type, including primitive types, void and array types.

⟨*type_pattern_expr*⟩ ::=
    ⟨*or_type_pattern_expr*⟩
  | ⟨*type_pattern_expr*⟩ '&&' ⟨*or_type_pattern_expr*⟩

⟨*or_type_pattern_expr*⟩ ::=
    ⟨*unary_type_pattern_expr*⟩
  | ⟨*or_type_pattern_expr*⟩ '||' ⟨*unary_type_pattern_expr*⟩

⟨*unary_type_pattern_expr*⟩ ::=
    ⟨*basic_type_pattern*⟩
  | '!' ⟨*unary_type_pattern_expr*⟩

⟨*basic_type_pattern*⟩ ::=
    'void'
  | ⟨*base_type_pattern*⟩
  | ⟨*base_type_pattern*⟩ ⟨*dims*⟩
  | '(' ⟨*type_pattern_expr*⟩ ')'

⟨*base_type_pattern*⟩ ::=
    ⟨*primitive_type*⟩
  | ⟨*name_pattern*⟩
  | ⟨*name_pattern*⟩ '+'

3.5.3 Class Name Pattern Expressions

This section defines the expressions that can be specified on class name patterns. Note that by *classname* we mean the name of any class, interface or aspect.

⟨*classname_pattern_expr_list*⟩ ::=
    ⟨*classname_pattern_expr*⟩
  | ⟨*classname_pattern_expr_list*⟩ ’,’ ⟨*classname_pattern_expr*⟩

⟨*classname_pattern_expr*⟩ ::=
    ⟨*and_classname_pattern_expr*⟩
  | ⟨*classname_pattern_expr*⟩ ’||’ ⟨*and_classname_pattern_expr*⟩

⟨*and_classname_pattern_expr*⟩ ::=
    ⟨*unary_classname_pattern_expr*⟩
  | ⟨*and_classname_pattern_expr*⟩ ’&&’ ⟨*unary_classname_pattern_expr*⟩

⟨*unary_classname_pattern_expr*⟩ ::=
    ⟨*basic_classname_pattern*⟩
  | ’!’ ⟨*unary_classname_pattern_expr*⟩

⟨*basic_classname_pattern*⟩ ::=
    ⟨*name_pattern*⟩
  | ⟨*name_pattern*⟩ ’+’
  | ’(’ ⟨*classname_pattern_expr*⟩ ’)’

⟨*classname_pattern_expr_nobang*⟩ ::=
    ⟨*and_classname_pattern_expr_nobang*⟩
  | ⟨*classname_pattern_expr_nobang ’——’ ¡and_classname_pattern_expr*⟩

⟨*and_classname_pattern_expr_nobang*⟩ ::=
    ⟨*basic_classname_pattern*⟩
  | ⟨*and_classname_pattern_expr_nobang*⟩ ’&&’ ⟨*unary_classname_pattern_expr*⟩

3.5.4 Method, Constructor and Field Patterns

⟨*method_constructor_pattern*⟩ ::=
  ⟨*method_pattern*⟩
| ⟨*constructor_pattern*⟩

⟨*method_pattern*⟩ ::=
  ⟨*modifier_pattern_expr*⟩ ⟨*type_pattern_expr*⟩ ⟨*classtype_dot_id*⟩
  '(' ⟨*formal_pattern_list_opt*⟩ ')' ⟨*throws_pattern_list_opt*⟩
| ⟨*type_pattern_expr*⟩ ⟨*classtype_dot_id*⟩
  '(' ⟨*formal_pattern_list_opt*⟩ ')' ⟨*throws_pattern_list_opt*⟩

⟨*constructor_pattern*⟩ ::=
  ⟨*modifier_pattern_expr*⟩ ⟨*classtype_dot_new*⟩
  '(' ⟨*formal_pattern_list_opt*⟩ ')' ⟨*throws_pattern_expr_opt*⟩
| ⟨*classtype_dot_new*⟩
  '(' ⟨*formal_pattern_list_opt*⟩ ')' ⟨*throws_pattern_expr_opt*⟩

⟨*field_pattern*⟩ ::=
  ⟨*modifier_pattern_expr*⟩ ⟨*type_pattern_expr*⟩ ⟨*classtype_dot_id*⟩
| ⟨*type_pattern_expr*⟩ ⟨*classtype_dot_id*⟩

3.5.5 Modifier Patterns

⟨*modifier_pattern_expr*⟩ ::=
  ⟨*modifier*⟩
| '!' ⟨*modifier*⟩
| ⟨*modifier_pattern_expr*⟩ ⟨*modifier*⟩
| ⟨*modifier_pattern_expr*⟩ '!' ⟨*modifier*⟩

3.5.6 Throws Patterns

⟨*throws_pattern_list_opt*⟩ ::=
  ϵ
| 'throws' ⟨*throws_pattern_list*⟩

⟨*throws_pattern_list*⟩ ::=
  ⟨*throws_pattern*⟩
| ⟨*throws_pattern_list*⟩ ',' ⟨*throws_pattern*⟩

⟨*throws_pattern*⟩ ::=
  ⟨*classname_pattern_expr_nobang*⟩
| '!' ⟨*classname_pattern_expr*⟩

3.5.7 Parameter List Patterns

Different levels of patterns are used for different formal patterns. In the following, the most general formal pattern is given, and then special restricted patterns that are used for pointcut parameters.

General parameter list patterns

⟨*formal_pattern_list_opt*⟩ ::=
    ϵ
  | ⟨*formal_pattern_list*⟩

⟨*formal_pattern_list*⟩ ::=
    ⟨*formal_pattern*⟩
  | ⟨*formal_pattern_list*⟩ ',' ⟨*formal_pattern*⟩

⟨*formal_pattern*⟩ ::=
    '..'
  | '.' '.'
  | ⟨*type_pattern_expr*⟩

Pointcut parameter list patterns

⟨*type_id_star_list_opt*⟩ ::=
    ϵ
  | ⟨*type_id_star_list*⟩

⟨*type_id_star_list*⟩ ::=
    ⟨*type_id_star*⟩
  | ⟨*type_id_star_list*⟩ ',' ⟨*type_id_star*⟩

⟨*type_id_star*⟩ ::=
    '*'
  | '..'
  | ⟨*type*⟩
  | ⟨*type*⟩ '+'

Acknowledgments

This work was supported, in part, by EPSRC and NSERC.

References

[1] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, 2003.