Parsing Challenges in Java 8

Erik Hogeman, Jesper Öqvist, Görel Hedin

Department of Computer Science

Lund University

JastAddJ

JastAddJ is a full source-to-bytecode modular Java compiler

- each Java version is a separate module
- Java 8 was implemented by Erik Hogeman for his Master's Thesis
- this talk is about the parsing challenges encountered

Java 8

Noteworthy features:

- Lambdas
- Method references
- Default methods

Lambdas

Java finally has anonymous functions!

```
(x, y) -> x + y

() -> { action1(); action2(); }
```

Lambda Example

Action listeners the old way:

```
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      print("hello");
   }
});
```

The new way, using lambda:

```
button.addActionListener( (e) -> print("hello") );
```

Method References

A way of using regular instance methods as lambdas:

```
Greeter greeter = new MyGreeter();
greetButton.addActionListener( greeter::greet );
exitButton.addActionListener( greeter::exit );
```

Default Methods

Interfaces can have non-abstract methods:

```
interface Greeter {
    default void greet(ActionEvent e) {
        print("greetings");
    }
    default void exit(ActionEvent e) {
        print("goodbye");
    }
}
class MyGreeter implements Greeter {
    // use default implementations
}
```

Parsing

We use an LALR parser for JastAddJ

- Generated with the Beaver parser generator
- Parser grammar is composed from parts in separate modules

Why an LR Parser Generator?

Advantages of a generated LR parser:

- Provably fast
- Generator certifies unambiguous grammar
- Decent tool support
- Bit more powerful than LL

Java 8 Parsing Challenges

- Ambiguous grammar specification
- Reduce-reduce conflicts between subexpressions
- Shift-reduce conflict
- Unlimited lookahead

Ambiguous Grammar Specification

Java spec (highly edited):

```
Expression -> Lambda
Expression -> ... -> Additive
   -> Multiplicative -> ... -> Cast
Cast -> (Type) Lambda
```

Input:

```
(T) (a, b) -> a * b;
```

Possible parse 1:

```
((T) (a, b) -> a) * b;
```

Possible parse 2:

```
(T) ((a, b) -> a * b);
```

The second one is desired. We achieved this by:

- changed the grammar
- lambda as primary expression
- lowered priority using precedence declarations

Lambda Reduce-Reduce Conflict

Lambda vs less-than expression:

```
(T<A> s) -> { } // lambda
(T<A)           // less-than expression
```

This is a reduce-reduce conflict.
Similar conflict in Java 5 with type cast:

```
(T<A>) s        // generic type cast
(T<A)           // less-than expression
```

In both cases the `T` terminal must be reduced to either *RelationalExpression* or *ReferenceType*.

Lambda Reduce-Reduce Conflict

We solved the reduce-reduce conflict by giving the related parsing productions explicit common prefixes:

```
Relational -> Name < Shift
Relational -> Relational < Shift
...
ReferenceType -> Name < TypeArguments_1
```

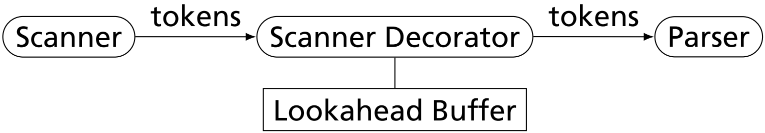
This removed the need to reduce the `Name` token too early.

Unlimited Lookahead

```
f(T<A, B>::m)    // method reference
f(T<A, B> m)     // less-than expression
```

There is no reasonable fixed lookahead that will allow the parser to decide between a less-than expression, or method reference.

Scanner Decorator



The Scanner Decorator looks ahead in the token stream when certain tokens are encountered, then potentially modifies the token stream.
In the previous case it inserts a synthetic `LT_TYPE` token.

Conclusions

- Java is not LR, but with some modifications we can make it LR(1)
- So far implemented nearly all of Java 8 features (parsing is complete)

Techniques we used to solve parsing challenges:

- Duplicate grammar to avoid reduce-reduce conflicts
- Introduce priority declarations to fix ambiguous grammar
- Scanner decorator to enable infinite lookahead

Questions!

Default Modifier Shift-Reduce

We parse all modifiers using the same production (for methods, interfaces, classes).
This introduced a shift-reduce conflict in switch-statements:

```
switch (x) {
case 0:
   default class A() { };
case 1:
   break;
default:
}
```

Intersection Type Cast

In Java 8 cast expressions can have the form:

```
(A & B & C) x
```

This form conflicts with binary expressions:

```
(A & B & C)
```

The conflict is very similar to the lambda versus less-than expression conflict.

Parsing Intersection Type Casts

We solve this conflict using the Scanner Decorator. Whenever a left-parenthesis is encountered, the decorator inserts the synthetic `INTERCAST` token if it determines that it is part of an intersection type cast.