IMPLEMENTING PARSERS AND STATE MACHINES IN JAVA

Terence Parr
University of San Francisco
Java VM Language Summit 2009
ISSUES

- Generated method size in parsers
- Why I need DFA in my parsers
- Implementing DFA in Java
- Predicated DFA edges
- My kingdom for dynamic scoping
- Exceptions for flow-control
- Computed gotos for interpreter instruction dispatch
Methods generated from big grammar rules can blow past 64k bytes (I’m trying to tighten up the generated code)

Solution: manually split rules into multiple; not obvious to most users

Can’t do automatically due to actions; issues with args/locals

```
A {...$x...}
| B {...$x...}
;
```

Won’t compile
Natural extension to LL(k) lookahead DFA: Allows cyclic DFA that can skip ahead past the modifiers to class or interface def

// LL(*), but non-LL(k) for any k
def : modifier* classDef
    | modifier* interfaceDef
    ;

Don’t approximate entire CFG with a regex; i.e., don’t include class or interface def rules

Predict and proceed normally with LL parse
SIMULATING STATE MACHINES

- Simulate DFA with bunch of tables

```java
public class T {
    static int[][] states
    static int[] s0 = {
        0, 0, 0, 2, 0, 0, 8, 0, 0, 0,
        0, 0, 0, 0, 1, 1, 0, 0, 1, 0,
        ...
    };
    static int[] s1 = { ... };
    ...
    static int[][] states = {s0, s1, ...};
}
```

No static arrays in .class: must init elements 1-by-1

```java
sipush n
newarray int ; create array
dup ; dups array
sipush i ; push index
iconst_0 ; value to store
iastore
```

5 or 6 bytes per element leaves room for only 10k elements for all tables in static ctor

aside from being slow to initialize, we run into the method size limit
To avoid static init issue, encode directly in Java.

Idea is to use CPU jmp instruction to change state. States are code addresses. Avoids big matrices, vectors.

“LR parsers can be made to run 6 to 10 times as fast as the best table-interpretive LR parsers.”*

But, can’t do arbitrary cyclic graphs w/o gotos in Java

Why not generate bytecodes directly?

because of predicated DFA edges; might have to compile arbitrary Java expressions (more in a second...)

* Thomas Pennello, Very Fast LR Parsing in Proceedings of the 1986 SIGPLAN symposium on Compiler construction
SO, WHAT DO WE DO?

- No gotos => must simulate DFA with arrays
- Encode shorts as chars: 0,9,32 is “\u0000\u0009\u0020”
- Encode arrays as strings, which are stored statically in the constant pool, to avoid static init size limit (got trick from jflex)
- Have to unpack into short/int arrays at runtime to initialize
- Run-length-encode to compress sparse matrices/arrays
Edges can be arbitrary expressions; can ref locals and parameters; can’t move predicates out of method() to predict()

```java
void method(boolean isAbstract) {
    methodHead();
    int alt = dfa39.predict(input);
    switch (alt) { ... }
}
```

Won’t compile in predict()
DYNAMIC SCOPING

- Idea: f() calls g(); g() can see f()'s parameters and locals
- Mostly evil, but solves some code gen issues:
  - let’s us automatically split large rule methods
  - let’s us move predicates out of context in generated DFA
- Or, I could manage my own parameter stack; can’t do that for locals, though (defined in arbitrary code)
- Backtracking parser must rewind upon failure and try next alternative

- IF-gates after every rule/token match is slow, big, messy

  ```
  alternative1
  if (!failed) return;
  rewind input
  alternative2
  ...
  ```

- But, aren’t exceptions very slow? Gafter told me only creating an exception object is slow; throwing is fast. I’m guessing faster than testing `failed` all the time
Overhead of fetch-decode-execute cycle switch/loop is high

Poor cache characteristics; perhaps even pipeline issues

Typical structure:

```c
while ( code[ip]! = HALT ) {
  switch ( code[ip] ) {
    case ADD : ... break;
    case JMP : ... break;
    case RET : ... break;
  }
  ip++;
}
```
Threaded interpreter puts dispatch into instruction implementation code; no loop; better cache characteristics

codeptr[] impl = { &ADD, &JMP, &RET, ... };

ADD: ... goto impl[code[+ip]];
JMP: ... goto impl[code[+ip]];
RET: ... goto impl[code[+ip]];

Dalvik VM trick: don’t even use address table; allocate n bytes per implementation where n is power of 2. Instr impl address is
&firstInstr+code[ip]<<lgn.
E.g., impl.’s at offsets 0, 16, 32, 48, ... for n=16
From language implementors point of view, would be nice to have:

- >64k bytecodes in methods
- static arrays in .class files
- gotos for DFA
- dynamic scoping (splitting rules, predicated DFA edges)
- computed gotos for interpreters

I’m not suggesting exposing all this to Java users

Perhaps secret option Neal Gafter quietly gives out? ;)

CONCLUSIONS