Daedalus Overview

Iavor S. Diatchki, Galois Inc.

November 2019

Overview

- Daedalus is a data description language
- Aimed at parsing formats with data dependencies
- Very much under construction

Parsers

- *Parsers* consume input, examine it, and if successful produce a *semantic value*.
- Parsers have names staring with a capital letter.

```
Digit      = '0' .. '9';
```

Character Classes

- A *character class* matches a single byte

```
0, '0'              -- match a specific byte
'a' | 'b'           -- union of character classes
'0' .. '9'          -- match a byte inthe range
! 'a'               -- complement
$byte               -- match any byte
```

- Character classes have names starting with $

```
$digit = '0' .. '9';
```

- Character classes can be used as parsers

```
HexDigit = $digit | 'A' .. 'F' | 'a' .. 'f';
```

Sequencing Parsers

Match `P`, then `Q`, then `R`:

```
{ P; Q; R }                 -- Result from `R`
{ $$ = P; Q; R }            -- Result from `P`
{ x  = P; Q; y = R }        -- Result is a record
```

Match a sequence of bytes:

```
"Hello"                     -- Result is an array of bytes
Many P                      -- Match multiple occurances of `P
```

Example:

```
HexNumber = { "0x"; Many<1..> HexDigit };
```

Alternatives

Choose between `P` and `Q` (same type):

```
P | Q      -- Result from `P` or `Q`, or both

P <| Q     -- Result from `P`, unless none, then `Q`
```

Choose between `P` and `Q` (different types):

```
Choose  { left = P; right = Q }      -- tagged union (all)

Choose1 { left = P; right = Q }      -- tagged union (first)
```

Data Dependency

Field values are in scope in later fields

```
{ len = Byte; body = Many<len> Byte }
```

Local variables

```
{ @len = Byte; Many<len> Byte }
```

Manipulating semantic values

```
{ @d = '0' .. '9'; ^ d - '0' }
```

Examining Semantic Values (1)

Records

```
Byte2 = { fst = Byte; snd = Byte };

Example = { @b = Byte2; ^ b.fst };
```

Unions

```
Sign    = Choose { pos = '+'; neg = '-' };

Example = { @s = Sign;
            @b = Byte;
            { s.pos; ^ b } | { s.neg; ^ 0 - b }
          };
```

Examining Semantic Values (2)

Booleans

```
Main = { $$ = Byte; ($$ < 128).true }
```

Arrays

```
Digit  = { @d = '0' .. '9'; ^ d as int };

Number = { @ds = Many<1..> Digit;
           ^ for (val = 0; d in ds) (val * 10 + d)
         };
```

Parameterized Parsers

Parameterized by a value

```
FixNum n        = Many<n> Digit;
```

Parameterized by another parser

```
Token P         = { $$ = P; Many ' ' };

Point           = { x = Token Number; y = Token Number };
```

Recursive Grammars/Types

Recursive grammars may result in recursive types

```
Value = Choose {
          number: Token Number;
          array:  Between "[" "]" Value;
        }

Between Open Close P = { Token Open; $$ = P; Token Close };
```

Challenges

- Error reporting
  - what is the location of a parse error?
  - errors for grammar developers vs. errors for users of generated parser
- Efficient parser generation
  - investigating using the algorithm from Yakker
- Can we avoid having a full-blown semantic value language?