DaeDaLus User Guide

Contents

DaeDaLus is a language for specifying parsers with data dependencies, which allows a parser's behavior to be affected by the semantic values parsed from other parts of the inputs. This allows a clear, yet precise, specification of many binary formats.

Declarations

A Daedalus specification consists of a sequence of *declarations*. Each declaration can specify either a *parser*, a *semantic value*, or a *character class*. Parsers may examine and consume input, and have the ability to fail. If successful, they produce a semantic value. Character classes describe sets of bytes, which may be used to define parsers, and will be discussed in more detail later.

The general form of a declarations is as follows:

```
def Name Parameters = Definition
```

The name of a declaration determines what sort of entity it defines:

- **parsers** always have names starting with an **uppercase** letter,
- **semantic values** have names starting with a **lowercase** letter,
- **character classes** have names starting with the symbol `$`.

Here are some sample declarations:

```
def P   = UInt8       -- a parser named `P`
def x   = true        -- a semantic value named `x`
def $d  = '0' .. '9'  -- a character class named `$d`
```

Single line comments are marked with `--`, while multi-line comment are enclosed between `{-` and `-}`, and may be nested.

Parsers

Primitive Parsers

Any Byte. The parser `UInt8` extracts a single byte from the input. It fails if there are no bytes left in the input. If successful, it constructs a value of type `uint 8`.

Specific Byte. Any numeric literal or character may be used as a parser. The resulting parser consumes a single byte from the input and succeeds if the byte matches the literal. Character literals match the bytes corresponding to their ASCII value.

Specific Byte Sequence. A string literal may be used to match a specific sequence of bytes. The resulting semantic value is an array of bytes, which has type `[uint 8]`.

End of Input. The parser `END` succeeds only if there is no more input to be parsed. If successful, the result is the trivial semantic value `{}`. Normally Daedalus parsers succeed as long as they match a *prefix* of the entire input. By

sequencing (section Sequencing Parsers) a parser with `END` we specify that the entire input must be matched.

Pure Parsers. Any semantic value may be turned into a parser that does not consume any input and always succeeds with the given result. To do so prefix the semantic value with the operator `^`. Thus, `^ 'A'` is a parser that always succeeds and produces byte `'A'` as a result.

Explicit Failure The `Fail` construct will always fail. This parser is parameterized by an optional location, along with an error message.

Examples:

```
{- Declaration                    Matches        Result          -}
def GetByte     = UInt8       -- Any byte X     X
def TheLetterA  = 'A'         -- Byte 65        65
def TheNumber3  = 3           -- Byte 3         3
def TheNumber16 = 0x10        -- Byte 16        16
def Magic       = "HELLO"     -- "HELLO"        [72,69,76,76,79]
def AlwaysA     = ^ 'A'       -- ""             65
def GiveUp      = Fail "I give up" -- Nothing    Failure with message "I give up"
```

Sequencing Parsers

Basic Sequencing. Multiple parsers may be executed one after the other, by listing them either between `{` and `}` or between `[` and `]`, and separating them with `;`. Thus, `{ P; Q; R }` and `[ P; Q; R ]` are both composite parsers that will execute `P` , then `Q`, and finally `R`. If any of the sequenced parsers fails, then the whole sequence fails.

Parsers sequenced with `[]` produce an array, with each element of the array containing the result of the corresponding parser. Since all elements in an array have the same type, all parsers sequenced with `[]` should construct the same type of semantic value.

By default, parsers sequenced with `{}` return the result of the last parser in the sequence.

Examples:

```
{- Declaration                       Matches        Result      -}
def ABC1 = { 'A'; 'B'; 'C' }      -- "ABC"          67
def ABC2 = [ 'A'; 'B'; 'C' ]      -- "ABC"          [65,66,67]
def ABC3 = { "Hello"; "ABC" }     -- "HelloABC"     [65,66,67]
def ABC4 = { "Hello"; 'C' }       -- "HelloC"       67
```

Explicit Result. A `{}`-sequenced group of parsers may return the result from any member of the group instead of the last one. To do so, assign the result of the parser to the special variable `$$`. For example, `{ P; $$ = Q; R }` specifies

that the group's result should come from `Q` instead of `R`. It is an error to assign `$$` more than once.

Local Variables. It is also possible to combine the results of some of the `{}`-sequenced parsers by using *local variables* and the pure parser. Assignments starting with the symbol `@` introduce a local variable, which is in scope in the following parsers. Here is an example:

```
def Add = {
  @x = UInt8;
  '+';
  @y = UInt8;
  ^ x + y
}
```

The parser `Add` is a sequence of 4 parsers. The local variables `x` and `y` store the results of the first and the third parser. The result of the sequence is the result of the last parser, which does not consume any input, but only constructs a semantic value by adding `x` and `y` together.

Structure Sequence. It is also possible to return results from more than one of the parsers in a `{}`-sequenced group. To do so give names to the desired results (*without* `@`). The semantic value of the resulting parser is a structure with fields containing the value of the correspondingly named parsers. Consider, for example, the following declaration:

```
def S = { x = UInt8; y = "HELLO" }
```

This declaration defines a parser named `S`, which will extract a byte followed by the sequence `"HELLO"`. The result of this parser is a *structure type*, also named `S`, which has two fields, `x` and `y`: `x` is a byte, while `y` is an array of bytes.

Note that structure fields also introduce a local variable with the same name, so later parsers in the sequence may depend on the semantic values in earlier parsers in the sequence. For example:

```
def S1 = { x = UInt8; y = { @z = UInt8; ^ x + z } }
```

The parser `S1` is a sequence of two parsers, whose semantic value is a structure with two fields, `x` and `y`. Both fields have type `uint 8`. The first parser just extracts a byte from input. The second parser is itself a sequence: first it extracts a byte from the input, but its semantic value is the sum of the two extracted bytes. As another example, here is an equivalent way to define the same parser:

```
def S2 = { x = UInt8; @z = UInt8; y = ^ x + z }
```

Syntactc Sugar. A number of the constructs described in this section are simply syntactic sugar for using local variables. Here are some examples:

Expression:	Equivalent to:
`{ $$ = P; Q }`	`{ @x = P; Q; ^ x }`

Expression:	Equivalent to:
`[ P; Q ]`	`{ @x0 = P; @x1 = Q; ^ [x0,x1] }`
`{ x = P; y = Q }`	`{ @x = P; @y = Q; ^ { x = x; y = y } }`

Parsing Alternatives

Biased Choice. Given two parsers `P` and `Q` we may construct the composite parser `P <| Q`. This parser succeeds if *either* `P` *or* `Q` succeeds. In the case that *both* succeed, the parser behaves like `P`. Note that `P` and `Q` have to construct semantic values of the same type.

More operationally, `P` would be used to parse the input first, and only if it fails would we execute `Q` on the same input. While this may be a useful intuition about the behavior of this parser, the actual parsing algorithm might implement this behavior in a different way.

Here are some examples:

```
{- Declaration            Matches        Result   -}
def B1 = 'A' <| 'B'    -- "A"            'A', or
                       -- "B"            'B'

def B2 = 'A' <| ^ 'B'  -- "A"            'A', or
                       -- ""             'B'
```

These two are quite different: `B1` will fail unless the next byte in the input is `'A'` or `'B'`, while `B2` never fails.

Unbiased Choice. Given two parsers `P` and `Q` we may construct the composite parser `P | Q`. This parser succeeds if either `P` or `Q` succeeds on the given input. Unlike biased choice, if *both* succeed, then the resulting parser is *ambigous* for the given input, which means that input may be parsed in more than one way. It is possible, however, to resolve ambiguities by composing (e.g., in sequence) with other parsers.

Here are some examples:

```
def U1 = 'A' | ^ 0
def U2 = { U1; 'B' }
```

Parser `U1` on its own is ambiguous on inputs starting with `"A"` because it could produce either `'A` (by consuming it from the input), or `0` (by consuming nothing). This happens because parsers only need to match a prefix of the input to succeed.

Parser `U2` accepts inputs starting with either `"AB"` (by using the left alternative of `U1`) or starting with `"B"` (by using the right alternative of `U1`). No inputs are ambiguous in this case.

Alternative Syntax. Given multiple parsers `A`, `B`, ... we can use the `Choose` keyword for unbiased choice and `Choose1` for biased choice.

Expression:	Equivalent to:
`Choose { A ; B; ...}`	`A | B | ...`
`Choose1 { A ; B; ... }`	`A <| B <| ...`

Choose can also be used to construct tagged unions: see below.

Control Structures

Guards

Simple boolean predicates, such as `<` or `==` may be used as a guard to control whether parsing continues. For example, the following parser uses the guard `(i - '0') > 5` to distinguish whether an parsed digit is greater than 5.

```
{
  @i = '0'..'9';
  Choose1 {
      { (i - '0') > 5; ^ "input gt 5";} ;
      { ^ "input leq 5";}
  }
}
```

Guards may also be made out of any boolean semantic value by using the `is` construct. So, if `p` is a boolean value, then `p is true` is a parser that succeeds without consuming input if `p` holds, and fails otherwise. Similarly, `p is false` is a parser that would succeed only if `p` is `false`.

Note that guard `x == y` is simply syntactic sugar for `(x == y) is true`.

For loops

The `for` construct can be used to iterate over collections (arrays and dictionaries). A for-loop declares a local variable representing the accumulated result of the computation, and a variable that is bound to the elements of the collection. The body may be a parser, or a semantic value. For example, the following expression sums the values in an array of integers:

```
for (val = 0 : int; v in [1,2,3])
  val + v
```

Here, `val` is initially bound to `0`. Each iteration of the loop binds `v` to the current element of the sequence, then computes the value of the body, `val + v`. This returned value is the updated value of `val`.

Another way to understand how this works is to see the following expression, which is the result of one step of evaluation:

```
for (val = 1; v in [2, 3])
  val + v
```

`for` supports an alternative form which binds both the index and value of a collection. For example, the following loop multiplies each element in the sequence by its index:

```
for (val = 0; i,v in [1,2,3])
  val + (i * v)
```

This construct is also useful when iterating over the contents of dictionaries, where the index is bound to the key. The following loop is a parser which fails when the value is less than the key:

```
for (val = 0; k,v in d)
  k <= v
```

Map

Daedalus supports another iteration construct, `map`. This performs an operation on each element of a sequence, resulting in a sequence of results. For example, the following code doubles each element in an array:

```
map (x in [1:int, 2, 3])
  2 * x
```

The `map` construct can be used to parse a sequence of blocks, based on a sequence of values. For example the following code parses blocks of the form `0AAA...`, with the number of `'A'` characters dicated by the input sequence.

```
map (x in [1, 2, 3]) {
  '0';
  Many x 'A';
}
```

Just as with `for`, the map construct has an alternative form that includes both sequence indexes and values:

```
map (i,x in [5, 2, 1]) {
  '0';
  len       = ^ { index = i, elem = x };
  something = Many x 'A';
}
```

Unions and Case Distinction

Daedalus supports tagged unions and case distinction on unions. The way to construct a union is to use `Choose`. For example, the following parser constructs a union with possible tags `good` and `bad`, depending on whether the input character is `'G'` or `'B'`.

```
Choose {
  good = 'G';
  bad = 'B';
}
```

It is also possible to construct a union literal using `{| good = 'G' |}`. Note however that the compiler will reject programs where it cannot infer the resulting type of the union. In such cases, you'd need to provide an explicit type signature.

Given a union `u` and tag name `t`, the guard `u is t` succeeds if the union has the correct tag. This can be used to control parser control flow, as in the following example:

```
{
  @res = Choose {
    good = 'G';
    bad = 'B';
  };
  Choose {
    {res is good; ^ "Success!"};
    {res is bad; ^ "Failure!"};
  }
}
```

The result of a succesful `is` guard is the value of the union element. For example

```
{
  @res = Choose {
    good = { 'G'; Many 'a' .. 'z' };
    bad =   'B' ;
  };
  Choose {
    { @msg = res is good; ^ (concat [ "Success!", msg])};
    { res is bad; ^ "Failure!" };
  }
}
```

Commit

Normally, at the point a parser fails, Daedalus will backtrack to a choice point and try an alternative parser. The `commit` guard acts as a cut-point and

prevents backtracking. For example, the following code cannot parse the string `"AC"` because parsing `'A'` and the subsequent `commit` will prevent backtracking reaching the alternative branch.

```
Choose1 {
  {'A'; commit; 'B' };
  {'A'; 'C' }  -- Can't happen
}
```

The `try` construct converts commit failure into parser failure. A commit failure will propagate until it hits an enclosing `try` construct, or until it escapes the top-level definition.

Option type

Daedalus supports the special polymorphic type maybe A, which has possible values nothing and just i, for some value of type A. The is guard can be used to identify which case holds.

```
{
  @res =
    {@l = 'A'..'Z'; ^ just l}
      <|
    {^ nothing};
  r = res is just
}
```

The above example could also be written using the builtin `Optional` parser.

```
{
  @res = Optional 'A'..'Z';
  r = res is just
}
```

Semantic Values

If successful, a parser produces a semantic value, which describes the input in some way useful to the application invoking the parser. In addition, semantic values may be used to control how other parts of the input are to be parsed. Daedalus has a number of built-in semantic values types, and allows for user-defined record and union types.

Booleans

The type `bool` classifies the usual boolean values `true` and `false`.

The operator `!` may be used to negate a boolean value.

Boolean values may be compared for equality using `==`.

Numeric Types

Daedalus supports a variety of numeric types: `int`, `uint N`, and `sint N`, the latter two being families of types indexed by a number. The type `int` classifies integers of arbitrary size. The `uint N` classify unsigned numbers that can be represented using `N` bits and `sint N` is for signed numbers that can be represented in `N` bits.

Literals of the numeric types may written either using decimal or hexadecimal notation (e.g., `10` or `0xA`). The type of a literal can be inferred from the context (e.g., `10` can be used as both `int` a `uint 8`).

Numeric types support basic arithmetic: addition, subtraction, multiplication, division, and modulus using the usual operators `+`,`-`,`*`,`/`, and `%`. DaeDaLus also supports shift operations `<<` and `>>`. These operations are overloaded and can be used on all numeric types, with the restriction that the inputs and the outputs must be of the same type.

Numeric types can also be compared for equality, using `==` and ordering using `<`, `<=`, `>`, and `>=`.

Unsigned integers may also be treated as bit-vectors, and support various bitwise operations: complement: `~`; exclusive-or `^`; and bitwise-and `&`. Unsigned numbers can also be appended to other numbers via the `<#` operator.

Stream manipulation

Daedalus parsers operate on an *input stream*, which by default is the input data to the parser. However, the input stream can be manipulated directly. For example, we can write a parser function which runs two different parsers on the same stream.

```
def ParseTwice P1 P2 = {
  @cur = GetStream;
  p1result = P1;
  SetStream cur;
  p2result = P2;
}
```

By manipulating the stream, we can also run a parser on a fixed-size sub-stream. The following parser parses a size-n chunk which begins with a sequence of letters, and then is filled with spaces:

```
def LetterFill n = {
  @cur = GetStream;
  @this = Take n cur;
  @next = Drop n cur;
  SetStream this;
  $$ = { $$ = Many {'A'..'Z'};
         Many ' ';
         END; };
  SetStream next;
}
```

It is also possible to directly access the current position in the stream using `Offset`. This can be used to calculate how many characters were read by a particular parser:

```
def OffsetTest = {
    a = Offset;
    "AA";
    b = Offset;
    "AAA";
    c = Offset;
}
-- Result: { a:0, b:2, c:5 }
```

The `arrayStream` operator converts an array into a stream:

```
def CatStream a b = {
    SetStream (arrayStream (concat [a, b]));
    "AA";
    "BBB";
    ^ {}
}
```

This example will succeed if the concatenation of the arrays `a` and `b` starts with the string `"AABBB"`.

Types

Character Classes

External Declarations