Deadalus User Guide

Contents

Declarations **1**

Parsers **2**
- Primitive Parsers 2
- Sequencing Parsers 3
- Parsing Alternatives 4

Semantic Values **5**
- Booleans 5
- Numeric Types 6

Types **6**

Character Classes **6**

External Declarations **6**

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*, separated by a semicolon. 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 parser succeed as long as they match a *prefix* of the entire input. By sequencing (Some test) 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.

Examples:

```
{- Declaration                    Matches        Result          -}
def GetByte     = UInt8       -- Any byte X     X
def TheLatterA  = '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
```

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, 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                   }`
`[ 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.

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, and multiplication, using the usual operators `+`, `-`, 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:

- Bitwise complement: `~`

Types

Character Classes

External Declarations