Rust コンパイラーウォークスルー

原 将己

2017 年 5 月 4 日

目次

第 1 章

はじめに

第 2 章

Rust コンパイラーの開発フロー

第 3 章

Rustc と Crate と Cargo

第 4 章

字句と構文

4.1 字句解析

Rust のソースファイルはまず字句解析にかけられます。

4.1.1 凡例

この節では BNF 風の記法を用いて文法を記述しています。BNF 風ですが実際には PEG (解析表現文法) に近い意味論です。

- `A B` は連接です。 2 つの構文要素をこの順に並べたものです。
- `A | B` は PEG の順序つき OR (`A / B`) です。 `A` を試し、失敗したら `B` を試します。ただし、この節で `A | B` と書いた場合は、順番による違いが生じないようになっているはずです。
- `A*`, `A+`, `A?` は、0 回以上の繰り返し、1 回以上の繰り返し、高々 1 回の出現です。いずれも貪欲に実行したものと解釈しますが、それを知らなくても読めるように配慮したつもりです。
- `lookahead A` と `not A` は先読み肯定と先読み否定です。慣れていないと読み間違えやすいかもしれません。 `A (not B)` は、 `A` だが `B` が後続しないものです。`(not B) A` は、 `A` だが同じ場所で `B` にはならないものです。
- `eof` は文字列の末尾です。
- "+=" のようにダブルクオートで囲んだものは文字列で、1 文字ずつ連接したものと同じです。C 言語風のエスケープを使っています。

4.1.2 Unicode

Rust の字句解析器は UTF-8 でエンコードされたファイルを入力します。UTF-8 をデコードしたものは、Unicode スカラー値の列とみなすことができます。

Unicode スカラー値とは、 `0` 以上 `0xD800` 未満か `0xE000` 以上 `0x110000` 未満の整数のことです。ここでは、Unicode 文字といったら Unicode スカラー値のことを指すことにします。

```
AnyChar ::= /* Any Unicode scalar value */
```

```
AnyAsciiChar ::=
  /*Any Unicode scalar value < 128 */
```

4.1.3　トークン列

Rust の字句解析器の仕事は、Unicode スカラー値の列を受け取り、字句エラーがなければトークン列を返すことです。

```
File ::= MaybeShebang Skip (Token Skip)* eof

Token ::=
    InnerDocComment
  | OuterDocComment
  | Symbol
  | Literal
  | Identifier
  | StrictKeyword
  | ReservedKeyword
  | Lifetime
```

- 字句的な特徴からトークンを分類すると、doc-comment, 記号、数値リテラル、文字列系リテラル、識別子とキーワード、生存期間と分けられます。しかし、構文解析器やマクロ展開器にとっての分類は必ずしもこれに沿った形ではありません。

4.1.4　空白とコメント

Rust の字句解析器は空白とコメントを無視します。コンパイラにとっては、これらはトークンを明示的に分割する以上の役割はありません。ただし、doc-comment は特別な扱いを受けます。

```
Skip ::= (Whitespace | Comment)* (not MaySkip)

Whitespace ::= PatternWhitespace+ (not PatternWhitespace)

PatternWhitespace ::=
  /* Any Unicode scalar value with PATTERN_WHITE_SPACE */

Comment ::= "//" (not ("/" | "!")) LineCommentChar* ("\n" | eof)
          | "////" LineCommentChar* ("\n" | eof)
          | "/*" (not ("*" | "!")) BlockCommentBody* "*/"
          | "/**/"

LineCommentChar ::= (not "\n") AnyChar
BlockCommentBody ::= (not ("/*" | "*/")) AnyChar
                   | "/*" BlockCommentBody* "*/"

MaySkip ::= PatternWhitespace
```

```
            | "//" (not ("/" | "!"))
            | "////"
            | "/*" (not ("*" | "!"))
            | "/**/"

MaybeShebang ::= Shebang | not Shebang

Shebang ::= "#!" (not "[") LineCommentChar* ("\n" | eof)
```

- UAX #31 [1] に従い、 Pattern_White_Space に分類される文字は全て空白文字とみなされます。 Pattern_White_Space は UAX #44 [2] で規定されており、 PropList.txt で取得できます。現時点では以下の文字が Pattern_White_Space に分類されます。
  - U+0009 Control: CHARACTER TABULATION
  - U+000A Control: LINE FEED (LF)
  - U+000B Control: LINE TABULATION
  - U+000C Control: FORM FEED (FF)
  - U+000D Control: CARRIAGE RETURN (CR)
  - U+0020 SPACE
  - U+0085 Control: NEXT LINE (NEL)
  - U+200E LEFT-TO-RIGHT MARK
  - U+200F RIGHT-TO-LEFT MARK
  - U+2028 LINE SEPARATOR
  - U+2029 PARAGRAPH SEPARATOR
- C と同様に // と /* がコメントとみなされますが、 doc-comment はトークンとみなされるため、この意味で通常のコメントとは異なります。
- C とは異なり、 /* はネストさせることができます。
- //// 型のコメントの中では、doc-comment と同様に、単独でのキャリッジリターンの出現が禁止されています。これはバグではないかと思います。
- 1 行目が #! で始まる場合、その行はシバンとみなされ、コメントと同様に無視されます。ただし、 #![で始まる場合は、内部属性と紛らわしいため、例外的にシバンとはみなされません。

4.1.5 doc-comment

doc-comment はコメントとよく似た構文を持ちますが、Rust コンパイラはこれを一つのトークンとみなします。

```
InnerDocComment ::=
    "//!" LineDocCommentChar* ("\r\n" | "\n" | eof)
  | "/*!" BlockDocCommentBody* "*/"

OuterDocComment ::=
    "///" (not "/") LineDocCommentChar* ("\r\n" | "\n" | eof)
```

```
  | "/**" (not "/") BlockDocCommentBody* "*/"

LineDocCommentChar ::= (not ("\r" | "\n")) AnyChar


BlockDocCommentBody ::= (not ("\r" | "/*" | "*/")) AnyChar
                      | "\r\n"
                      | "/*" BlockDocCommentBody* "*/"
```

- ///, //!, /**, /*! で始まるコメントは、doc-comment と解釈されます。
- 例外として、 //// で始まるコメントと /**/ は、doc-comment ではありません。
- これらの doc-comment は、外部ツールだけではなく、Rust コンパイラによって認識されます。字句的に正しい doc-comment であっても、構文的に誤った位置にあれば、コンパイルエラーになる可能性があります。
- 通常のコメントと異なり、doc-comment 内ではキャリッジリターン (CR) が単独で出現してはいけません。必ずラインフィードと対で (CRLF) 出現する必要があります。
- C とは異なり、 /* はネストさせることができます。

4.1.6　記号

```
Symbol ::=
    "=" (not ("=" | ">"))
  | "<" (not ("=" | "<" | "-"))
  | "<=" | "==" | "!=" | ">="
  | ">" (not ("=" | ">"))
  | "&&" | "||"
  | "!" (not "=")
  | "~"
  | "+" (not "=")
  | "-" (not ("=" | ">"))
  | "*" (not "=")
  | "/" (not "=")
  | "^" (not "=")
  | "&" (not ("=" | "&"))
  | "|" (not ("=" | "|"))
  | ">>" (not "=")
  | "<<" (not "=")
  | "+=" | "-=" | "*=" | "/=" | "^="
  | "&=" | "|=" | ">>=" | "<<="
  | "@"
  | "." (not ".")
  | ".." (not ".")
  | "..." | "," | ";"
  | ":" (not ":")
  | "::" | "->" | "<-" | "=>" | "#"
```

```
  | "$" | "?"
  | "(" | ")" | "{" | "}" | "[" | "]"
  | Underscore
```

- 記号のパース規則は単純です。記号の候補の中で、一番長いものを選択します。
- アンダースコアだけ特殊です。アンダースコアは字句的には識別子に近いですが、パーサーに渡す段階では識別子よりも記号に近い扱いを受けます。

4.1.7 リテラル

```
Literal ::= LiteralBody LiteralSuffix?

LiteralBody ::= NumberLiteralBody | StringLikeLiteralBody

LiteralSuffix ::= IdentifierOrKeyword | Underscore
```

- リテラルは、数値リテラルと、文字列系リテラルの 2 種類あります。
- いずれのリテラルにも、識別子またはキーワードを後置することができます。これは型を表すのに使います。
- 識別子またはキーワードのかわりにアンダースコアを後置することもできます。リテラルの構造上数値リテラルでは不可能です。後置しても、パーサーからはなかったものとして扱われます。もしかしたらバグかもしれません。
- 後置された文字列は型として解釈されますが、これは構文解析器の仕事です。

4.1.8 数値リテラル

```
NumberLiteralBody ::= IntegerLiteralBody | FloatLiteralBody

IntegerLiteralBody ::=
    "0b" "_"* BinDigit BinDigitU* (not DecDigitU) EnsureNotFloat
  | "0o" "_"* OctDigit OctDigitU* (not DecDigitU) EnsureNotFloat
  | Decimal (not DecDigitU) EnsureNotFloat
  | "0x" "_"* HexDigit HexDigitU* (not HexDigitU) EnsureNotFloat

EnsureNotFloat ::= not ("." | "e" | "E")
                 | lookahead ".."
                 | lookahead ("." IdentStart)

FloatLiteralBody ::=
    Decimal "." (not ("." | IdentStart | DecDigit))
  | Decimal "." Decimal (not ("e" | "E"))
  | Decimal "." Decimal ExponentPart
  | Decimal ExponentPart

ExponentPart ::=
    ("e" | "E") ("-" | "+")?
```

```
21      "_"* DecDigit DecDigitU* (not DecDigitU)
22
23  Decimal ::= DecDigit DecDigitU*
24
25  BinDigit ::= "0" | "1"
26
27  OctDigit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
28
29  DecDigit ::= OctDigit | "8" | "9"
30
31  HexDigit ::=
32      DecDigit | "A" | "B" | "C" | "D" | "E" | "F"
33    | "a" | "b" | "c" | "d" | "e" | "f"
34
35  BinDigitU ::= BinDigit | "_"
36
37  OctDigitU ::= OctDigit | "_"
38
39  DecDigitU ::= DecDigit | "_"
40
41  HexDigitU ::= HexDigit | "_"
```

- 数値リテラルは、整数リテラルと浮動小数点数リテラルからなります。
- 負の符号は単なる単項演算子です。トークンとしては、非負の数のみ扱われます。
- アンダースコアで数を区切ることができます。しかし、数字がないのにアンダースコアだけある、という状況は禁止されています。また、リテラルの先頭や、小数点の直後には、アンダースコアを置くことはできません。
- 2 進数、8 進数、16 進数では接頭辞を明示する必要があります。`0` で始まっていても 10 進数です。また、接頭辞は小文字でなければなりません。16 進数の桁や、指数部をあらわす `e` は大文字でも小文字でも構いません。
- 小数点の手前を省略することはできません。また、小数点の直後を省略できるのは、直後が `.` でも識別子でもない場合だけです。
- 整数の直後に `.` を置けるのは、その直後が `.` または識別子の場合だけです。
- 上記の規則について、現在は `._` が特別扱いされています。おそらくバグではないかと思います。
- また、現在は `0e`, `0E` から始まる浮動小数点数がパースエラーになるバグがあります。これは修正済みで、次の安定板には入ると思います。
- `a` から `f` までの文字は、16 進数以外では、トークン区切りとみなされます。しかし、たとえ 8 進数であっても、`9` はトークン区切りとはみなされません。この位置でパースエラーになります。
- `f` という文字は、微妙なバランスの上に成立しています。16 進数は整数でしか使えないので、接尾辞には `i` か `u` が指定できれば十分です。いっぽう、`f` を含む接尾辞が指定される浮動小数点数では、`f` の部分でリテラル本体のパースが打ち切られるため、うまく `f` を指定することができます。

4.1.9 文字列系リテラル

```
StringLikeLiteralBody ::=
    ByteLiteralBody | ByteStringLiteralBody
  | CharLiteralBody | CharStringLiteralBody

ByteLiteralBody ::=
    "b\'" (not ("\\" | "\t" | "\n" | "\r" | "\'")) AnyAsciiChar "\'"
  | "b\'\\" ("n" | "r" | "t" | "\\" | "\'" | "\"" | "0") "\'"
  | "b\'\\x" HexDigit HexDigit "\'"

ByteStringLiteralBody ::=
    "b\"" SingleByte* "\""
  | "br" repeated("#", N) "\""
      ((not ("\"" repeated("#", N))) AnyAsciiChar)*
      "\"" repeated("#", N)
      /* N >= 0 */
      /* repeated("#", 3) means e.g. "###" */

SingleByte ::=
    (not ("\\" | "\r" | "\"")) AnyAsciiChar
  | "\r\n"
  | "\\" ("n" | "r" | "t" | "\\" | "\'" | "\"" | "0")
  | "\\" ("\r\n" | "\n") Whitespace
  | "\\x" HexDigit HexDigit

CharLiteralBody ::=
    "\'" (not ("\\" | "\t" | "\n" | "\r" | "\'")) AnyChar "\'"
  | "\'\\" ("n" | "r" | "t" | "\\" | "\'" | "\"" | "0") "\'"
  | "\'\\x" OctDigit HexDigit "\'"
  | "\'\\u{" HexDigit{1, 6} "}\'"
    /* only when x < 0xD800 or 0xE000 < x < 0x110000 */

CharStringLiteralBody ::=
    "\"" SingleChar* "\""
  | "r" repeated("#", N) "\""
      ((not ("\"" repeated("#", N))) AnyChar)*
      "\"" repeated("#", N)
      /* N >= 0 */
      /* repeated("#", 3) means e.g. "###" */

SingleChar ::=
    (not ("\\" | "\r" | "\"")) AnyChar
  | "\r\n"
  | "\\" ("n" | "r" | "t" | "\\" | "\'" | "\"" | "0")
  | "\\" ("\r\n" | "\n") Whitespace
```

```
  | "\\x" OctDigit HexDigit
  | "\\u{" HexDigit{1, 6} "}"
    /* only when x < 0xD800 or 0xE000 < x < 0x110000 */
```

- バイト、バイト列、文字、文字列のいずれかを表します。
- バイトまたは文字は ' で囲み、バイト列または文字列は " で囲みます。
- バイトまたはバイト列をあらわすときは b を前置します。
- バイト列または文字列の場合は生リテラルを使うこともできます。生リテラルは " のかわりに r###" のような形で開始します。この場合最初に "### が出現した位置で終了になります。 # の個数は 0 個以上で、左右で揃える必要があります。
- バックスラッシュ、水平タブ、ラインフィード、キャリッジリターン、デリミタ (" または ' のうちリテラルを囲うのに使われているほう) 以外の ASCII 文字は、そのまま書くことができます。
- ASCII 以外の Unicode 文字 (Unicode スカラー値) は、文字または文字列リテラルにそのまま書くことができます。
- タブ、ラインフィード、CRLF (キャリッジリターン + ラインフィード) は、生リテラルでないバイト列または文字列に直接書くことができます。CRLF は LF とみなされます。
- 生リテラル以外のリテラルでは、 \n, \r, \t, \\, \', \", \0 がエスケープとして利用できます。
- 生リテラル以外のリテラルでは、 \x に続けて 16 進法で 2 桁 (アンダースコア不可) の整数を指定することで、1 バイトエスケープになります。ただし、 128 より大きいバイトは、文字または文字列リテラルでは指定できません。
- 生リテラル以外の文字または文字列リテラルでは、 \u{} の中に 16 進法で 1 桁から 6 桁まで (アンダースコア不可) の整数を指定することで、Unicode エスケープになります。ただし指定できるのは Unicode スカラー値、つまり 0 以上 0xD800 未満か、 0xE000 以上 0x110000 未満の整数のみです。
- 生リテラル以外のバイト列または文字列リテラルでは、バックスラッシュの直後に改行 (LF または CRLF) を続けると、該当バックスラッシュから、連続する空白文字が全てスキップされます。
- 生リテラルではエスケープは使用できず、全ての文字がそのまま解釈されます。生バイト列リテラルでは、ASCII 文字以外 (128 以上のスカラー値) は使用できません。

4.1.10　識別子またはキーワード

```
Identifier ::=
  (not (StrictKeyword | ReservedKeyword)) IdentifierOrKeyword

IdentifierOrKeyword ::=
  (not (Underscore | MaybeString))
  IdentStart IdentContinue* (not IdentContinue)

```

```
IdentStart ::=
  /* Any Unicode scalar value with XID_Start */ | "_"

IdentContinue ::=
  /* Any Unicode scalar value with XID_Continue */ | "_"

Underscore ::= "_" (not IdentContinue)

MaybeString ::= "r\"" | "r#" | "b\"" | "b\'" | "br\"" | "br#"

StrictKeyword ::= StrictKeywordString (not IdentContinue)

StrictKeywordString ::=
    "as" | "box" | "break" | "const" | "continue" | "crate"
  | "else" | "enum" | "extern" | "false" | "fn" | "for"
  | "if" | "impl" | "in" | "let" | "loop" | "match" | "mod"
  | "move" | "mut" | "pub" | "ref" | "return" | "self" | "Self"
  | "static" | "struct" | "super" | "trait" | "true" | "type"
  | "unsafe" | "use" | "where" | "while"

ReservedKeyword ::= ReservedKeywordString (not IdentContinue)

ReservedKeywordString ::=
    "abstract" | "alignof" | "become" | "do" | "final"
  | "macro" | "offsetof" | "override" | "priv" | "proc"
  | "pure" | "sizeof" | "typeof" | "unsized" | "virtual"
  | "yield"

WeakKeyword ::= WeakKeywordString (not IdentContinue)

WeakKeywordString ::= "default" | "union"
```

- 識別子またはキーワードは、 ASCII の数字、大文字、小文字、アンダースコアからなります。また最初の文字は ASCII の大文字、小文字、アンダースコアのいずれかでなければなりません。
- この規則は `feature(non_ascii_idents)` により Unicode に拡張されます。この場合、識別子は `XID_Continue` な文字またはアンダースコアからなります。また、最初の文字は `XID_Start` な文字またはアンダースコアでなければなりません。これは UAX #31 [1] を考慮していますが、現在の実装は完全とはいえません。UAX #31 では NKFC 等の正規化が言及されていますが、現在のコンパイラは正規化を全くしません。どの正規化をするかなどを含めて、議論の途中の状態です。
- アンダースコア 1 文字からなる場合は、識別子ではなく記号として解釈されます。
- `r"`, `r#`, `b"`, `b'`, `br"`, `br#` のいずれかで始まる場合は、これらの `r`, `b`, `br` は識別子ではなく文字列系リテラルの開始として解釈されます。
- 「識別子またはキーワード」のうち、いくつかは「強キーワード (strict keyword)」「予約キーワード (reserved keyword)」「弱キーワード (weak keyword)」のいずれか

に分類されています。強キーワードは特別な意味をもつもので、予約キーワードは将来強キーワードとして使われる可能性を考慮して予約されているものです。

- 強キーワードでも予約キーワードでもないものを識別子といいます。弱キーワードは、原則として識別子として扱われますが、文脈によっては特別な意味を持ちます。
- 識別子またはキーワードは、マクロの観点からはキーワードか否かに関係なく同等に扱われます。識別子またはキーワードと、生存期間には、構文文脈が付与されます。アンダースコアは識別子またはキーワードではないため、構文文脈は付与されません。
- 字句解析器は、空の構文文脈を付与します。

4.1.11　生存期間

```
Lifetime ::=
    (not "\'" IdentStart "\'")
    "\'" IdentifierOrKeyword

WeakLifetimeKeyword ::= "\'static" (not IdentContinue)
```

- ' に識別子またはキーワードを後続させたものは、生存期間です。
- ただし、文字リテラルとしても解釈できる場合は、文字リテラルが優先されます。
- '_ は生存期間ではありません。
- 'static は生存期間ですが、文脈によっては特別な意味を持ちます。
- 識別子またはキーワードと、生存期間には、構文文脈が付与されます。
- 字句解析器は、空の構文文脈を付与します。

4.1.12　特殊なトークン

字句解析器以外によってのみ生成される特殊なトークンがいくつかあります。

- 空文字列、 {{root}}, $crate はいずれも、上で説明した「識別子またはキーワード」の条件を満たしていませんが、内部的に識別子として生成される場合があります。
- 同じく、空文字列が内部的に生存期間として生成される場合があります。なお、通常の生存期間の名前は、内部的には ' を含んだ形で保存されています。
- 補間トークンは、AST を 1 つのトークンとしてみなしたものです。
  - アイテム、ブロック、文、パターン、式、型、識別子を含む補間トークンは、マクロの展開のために使われます。
  - メタアイテム、パス、トークンツリーを含む補間トークンは、属性構文拡張の展開のために使われます。
  - パターンマッチの腕、実装内アイテム、トレイト内アイテム、ジェネリックス、where 節、関数の仮引数部分は、準クオートの実現のために内部的に使用されます。
- マクロの変数置換 $x を表すためのトークン。内部的には $ を含まない形で保存さ

れます。構文文脈も保存されます。

4.2 構文解析

Rust の構文解析器は、大まかにいうと次の 2 つの仕事を行います。

- トークン列を抽象構文木に変換する。
- `mod foo;` がある場合は、別のファイルを新規に読み出して再帰的に字句解析と構文解析を呼び出す。

後者は、C 言語でいうところの `#include` と似ていると考えてよいでしょう。ただし、Rust ではプリプロセッサーではなく構文解析器がこれを行います。

4.2.1 構文解析のための字句

Rust のトークン列は、構文解析器から見ると大雑把に以下のように分類されます。

ここでは簡単のために、記号やキーワードなどのトークンは単に `"*="` や `"fn"` といった文字列として表記します。実際には単一のトークンにマッチすると考えてください。

```
Token ::= NonParen | "(" | ")" | "{" | "}" | "[" | "]"

NonParen ::=
    InnerDocComment
  | OuterDocComment
  | "=" | "<" | "<=" | "==" | "!=" | ">=" | ">"
  | "&&" | "||" | "!" | "~"
  | "+" | "-" | "*" | "/" | "^" | "&" | "|" | ">>" | "<<"
  | "+=" | "-=" | "*=" | "/=" | "^="
  | "&=" | "|=" | ">>=" | "<<="
  | "@" | "." | ".." | "..." | "," | ";" | ":"
  | "::" | "->" | "<-" | "=>" | "#" | "$" | "?"
  | "_"
  | IntegerLiteral | FloatLiteral
  | ByteLiteral | ByteStringLiteral
  | CharLiteral | CharStringLiteral
  | Identifier
  | Lifetime
  | "as" | "box" | "break" | "const" | "continue" | "crate"
  | "else" | "enum" | "extern" | "false" | "fn" | "for"
  | "if" | "impl" | "in" | "let" | "loop" | "match" | "mod"
  | "move" | "mut" | "pub" | "ref" | "return" | "self" | "Self"
  | "static" | "struct" | "super" | "trait" | "true" | "type"
  | "unsafe" | "use" | "where" | "while"
  | "" | "{{root}}" | "$crate"
  | NtItem | NtBlock | NtStmt | NtPat | NtExpr | NtTy | NtIdent
  | NtMeta | NtPath | NtTT
  | NtArm | NtImplItem | NtTraitItem | NtGenerics
```

```
  | NtWhereClause | NtArg
  | SubstNt
```

- 記号や強キーワードはそれぞれ別種のトークンとして扱われます。
- 弱キーワード default, union, 'static は原則として識別子や生存期間の一種として扱われますが、特定の文脈では特別扱いされます。
- リテラルの種類は構文解析にとってはそれほど重要ではありませんが、整数リテラルと文字列リテラルは特別な扱いを受けることがあります。

4.2.2　列の文法

Rust の構文では、コンマ区切りの列が様々な場所で出てきます。これらの多くは、0 個以上の要素が指定可能で、末尾に最大 1 個の余分なコンマをつけることが許されています。

この節では、このパターンを、以下のように定義される sequence(E, D) を用いて略記します。

```
sequence(E, D) ::= (E (D E)* D?)?
```

4.2.3　crate

```
Crate ::= InnerAttribute* Item*
```

- crate 全体は mod とほぼ同じですが、構造上外部属性が指定できません。

4.2.4　アイテム

```
Item ::= OuterAttribute* (NtItem | Visibility? ItemBody)

ItemBody ::=
    ItemExternCrate
  | ItemUse
  | ItemStatic
  | ItemConst
  | ItemFn
  | ItemMod
  | ItemForeign
  | ItemType
  | ItemEnum
  | ItemStruct
  | ItemUnion
  | ItemTrait
  | ItemImpl
  | ItemMacro
```

- 関数や型など、Rust でグローバルな位置に置けるものはアイテムと呼ばれます。

- アイテムには属性と可視性を付与することができます。内部属性を書けるアイテムについては、内部属性と外部属性はマージされます。
- 構文解析済みのトークン (NtItem) が来た場合はそれをそのまま受理します。NtItem には可視性を付与することはできません。 NtItem に由来する属性と、NtItem に付与した外部属性は、マージされます。
- `unsafe` や `extern` などのキーワードを前置できるアイテムがあるため、アイテムの種類を決定するには数トークンほど先読みをする必要があります。
- アイテム (トレイト内アイテム、実装内アイテム、外部アイテムを含む) の構文上の特徴として、波括弧もしくはセミコロンで終端されるというものがあります。なお、セミコロン単体はアイテムではありません。
- アイテム (トレイト内アイテム、実装内アイテム、外部アイテムを含む) のもう一つの構文上の特徴として、マクロ以外はキーワード (弱キーワードを含む) から開始されるというものがあります。

4.2.5 外部 crate 宣言

```
ItemExternCrate ::=
  "extern" "crate" Identifier ("as" Identifier)? ";"
```

- crate のルートモジュールを `as` で指定した名前で参照します。
- `as` を省略した場合は、crate 名が使われます。

4.2.6 インポート

```
ItemUse ::=
    "use" ModPath ("as" Identifier)? ";"
  | "use" ModPath? "::" "*" ";"
  | "use" (ModPath? "::")? "{" sequence(UseListElement, ",") "}" ";"

UseListElement ::= (Identifier | "self") ("as" Identifier)?
```

- 定義に対する別名づけを行います。単独インポートとグロブインポートの 2 種類があります。
- `{ }` で囲んだものはリストインポートといい、複数の単独インポートに展開されます。 `self` を使うことで、モジュールそのものとモジュール内の定義を同時にインポートすることもできます。
- 単独インポートで `as` を省略した場合は、パスの最後の要素が採用されます。
- パスが通常の識別子から始まる場合は、強制的に絶対パスとして解釈されます。

4.2.7 静的アイテムと定数アイテム

```
ItemStatic ::= "static" "mut"? Identifier ":" Type "=" Expression ";"

ItemConst ::= "const" Identifier ":" Type "=" Expression ";"
```

- 定数または静的記憶期間を持つ領域を定義します。定数に mut はつけられません。
- let とは異なり、型は必須です。また、let のように左辺にパターンは使えません。

4.2.8　関数定義

```
ItemFn ::=
  ItemFnPrefix "fn" Identifier Generics?
  "(" sequence(Pattern ":" Type, ",") ")" ("->" Type)?
  WhereClauses?
  Block

ItemFnPrefix ::=
    "unsafe"? ("extern" CharStringLiteral?)?
  | "const" "unsafe"?
```

- 関数を定義します。
- feature(const_fn) により const が指定できます。指定すると、定数文脈でその関数を利用できるようになります。
- unsafe を指定すると、この関数の利用は unsafe だとマークされます。同時に、関数内部では unsafe 機能の利用が可能になります。
- extern により ABI を指定できます。 extern を省略すると "rust" ABI になります。 extern とだけ指定すると "C" ABI になります。
- const と extern は同時指定できません。
- ジェネリックスおよび where が指定できます。ジェネリックスの位置は関数名の直後で、 where の位置は戻り値型の直後です。
- 引数リストはパターンと型を並べたものです。他の fn に比べて最も単純といえます。
- ブロック内に内部属性があれば、それは関数の属性になります。

4.2.9　モジュール

```
ItemMod ::=
    "mod" Identifier ";"  /* Parser treats it specially. */
  | "mod" Identifier "{" InnerAttribute* Item* "}"

ModuleFile ::= InnerAttribute* Item*
```

- mod foo {} はモジュールを定義します。中には内部属性とアイテムを書くことができます。
- mod foo; のようにセミコロン終端された mod は、それ自身はモジュールの中身を持っていません。そこで、構文解析器はその場で別のファイルを開いて、そこからモジュールの中身を読み取ります。
- mod foo; の処理は、このアイテムに付属している外部属性に影響を受けます。ファイルを発見するまでは内部属性は入手できませんから、ファイル発見にかかわ

る設定は外部属性として書かなければいけません。

- 通常、属性の内容は構文解析より後に処理されますが、 `mod` に付与された `cfg_attr(...)` はその場で処理されます。(`mod foo {}` でも処理されますが、これが仕様かどうかはわかりません。)
- さらに、 `mod foo;` 形式の場合は、 `cfg(...)` がその場で処理されます。偽と評価された場合はファイルの探索を行いません。 (`mod foo {}` では処理されませんが、これが仕様かどうかはわかりません。)
- `mod foo;` のファイル発見規則はやや複雑です。まず、ローカル crate の各モジュールにはディレクトリが紐付けられています。また、そのモジュールがディレクトリを「所有しているかどうか」というフラグも管理されています。これに基いて以下のようにファイルが発見されます。
  - crate のルートモジュールのファイルとディレクトリは、コンパイラドライバによって指定される。例えば `src/main.rs` と `src` / 所有 が指定される。
  - `mod foo {}` は親モジュールのディレクトリの直下の `foo` が紐付けられる。所有関係は親モジュールから継承される。
  - ただし、 `#[path = "somewhere"]` (ディレクトリ) が指定された場合は、親モジュールからの相対パスでこのディレクトリが紐付けられ、このディレクトリを所有していることになる。
  - ブロックには、それを含むモジュールと同じディレクトリが紐付けられるが、そのディレクトリを所有はしない。
  - `mod foo;` はまずファイルを探索する。親モジュールのディレクトリの直下の `foo.rs` と `foo/mod.rs` のうち、存在しているほうを採用する。どちらもなかったり、どちらも存在している場合はエラーとなる。また、親モジュールがディレクトリを所有していない場合はエラーとなる。
  - ただし、 `#[path = "somewhere"]` (ファイル) が指定された場合は、上記のファイルを探索せずに、指定されたファイルを採用する。親モジュールがディレクトリを所有していなくてもエラーとはならない。
  - 採用されたファイルの直上のディレクトリが紐付けられる。ファイル名が `mod.rs` だった場合のみ、所有しているものとみなされる。
  - マクロ呼び出しの実引数がパースされるさいは、マクロ呼び出しの位置でパースされたかのように振る舞う。マクロ展開結果がパースされるさいも、マクロ呼び出しの位置でパースされたかのように振る舞う。

4.2.10 外部アイテムの参照

```
1 ItemForeign ::=
2   "extern" CharStringLiteral?
3   "{" InnerAttribute* ForeignItem* "}"
4
5 ForeignItem ::=
6   OuterAttribute* Visibility?
7   (ForeignItemStatic | ForeignItemFn)
```

```
8
9  ForeignItemStatic ::=
10   "static" "mut"? Identifier ":" Type ";"
11
12 ForeignItemFn ::=
13   "fn" Identifier Generics?
14   ForeignItemFnArgs ("->" Type)?
15   WhereClauses? ";"
16
17 ForeignItemFnArgs ::=
18     "(" sequence(Pattern ":" Type, ",") ")"
19   | "(" (Pattern ":" Type ",")+ "..." ")"
```

- 外部で定義されたシンボルをアイテムとして参照するには `extern { }` で囲みます。ABI 名を省略した場合は `"C"` として扱われます。
- 静的変数や関数の定義と似ていますが、定義部分がありません。
- `fn` に `const`, `unsafe`, `extern` などを追加で指定することはできません。
- 個別のアイテムに可視性をつけることができます。なお、 `extern` 自身に可視性をつけても構文解析後にエラーになります。
- 関数定義と同様、引数名は必要です。引数はパターンとしてパースされますが、識別子 (`mut` なし) とワイルドカード以外のパターンを指定すると構文解析後にエラーになります。
- 引数リストの末尾に `...` を指定することで可変長引数を表現できます。可変長部分の前に 1 つ以上の引数が必要です。

4.2.11　型別名

```
1  ItemType ::=
2    "type" Identifier Generics? WhereClauses? "=" Type ";"
```

- 型別名を定義します。型別名は、型検査では元の型とほぼ同じものとして扱われます。

4.2.12　列挙型

```
1  ItemEnum ::=
2    "enum" Identifier Generics? WhereClauses?
3    "{" (EnumBody | EnumBodyWithDiscriminator) "}"
4
5  EnumBody ::=
6    sequence(
7      RecordEnumVariant | TupleEnumVariant | UnitEnumVariant,
8      ",")
9
10 EnumBodyWithDiscriminator ::=
```

```
  sequence(
    UnitEnumVariant | EnumBodyWithDiscriminator,
    ",")

RecordEnumVariant ::=
  OuterAttribute* Identifier "{" RecordStructBody "}"

TupleEnumVariant ::=
  OuterAttribute* Identifier "(" TupleStructBody ")"

UnitEnumVariant ::=
  OuterAttribute* Identifier

EnumVariantWithDiscriminator ::=
  OuterAttribute* Identifier "=" Expression
```

- 代数的データ型を定義します。代数的データ型の各腕はバリアントと呼ばれます。
- バリアントの定義は構造体とほぼ同様です。ユニット形式、タプル形式、レコード形式があります。
- ユニット形式のバリアントのみからなる列挙型は C 風列挙型 (C-like enum) と呼ばれます。C 風列挙型は C 言語の列挙型とほぼ同様です。
- 特別なバリアントとして、判別子 (discriminator) を明示するバリアントがあります。判別子の明示は C 風列挙体でのみ使えます。この判定は構文解析の途中に行われますが、これが仕様かどうかはよくわかりません。

4.2.13 構造体

```
ItemStruct ::=
  "struct" Identifier Generics?
  ( WhereClauses? ";"
  | WhereClauses? "{" RecordStructBody "}"
  | "(" TupleStructBody ")" WhereClauses? ";")

RecordStructBody ::= sequence(StructDeclField, ",")

StructDeclField ::=
  OuterAttribute* Visibility? Identifier ":" Type

TupleStructBody ::=
  sequence(OuterAttribute* Visibility? Type, ",")
```

- 構造体型を定義します。C 言語と同様の、複数のフィールドを横に並べてできる型です。
- 構造体はその形式により 3 種類に分けられます。ユニット構造体、タプル構造体、レコード構造体です。
- ユニット構造体はフィールドリストを持ちません。したがってフィールドは必ず 0

個です。

- タプル構造体は丸括弧で囲まれたフィールドリストを持ちます。各フィールドは名前を持たず、0,1,2,... という番号で識別されます。
- レコード構造体は波括弧で囲まれたフィールドリストを持ちます。各フィールドは名前を持ちます。

4.2.14　共用体

```
ItemUnion ::=
  "union" Identifier Generics? WhereClauses?
  "{" RecordStructBody "}"
```

- 共用体型を定義します。C 言語と同様の、複数のフィールドを同じ先頭アドレスに割り当ててできる型です。
- 構文上は、共用体はレコード構造体とほぼ同じです。
- union は弱キーワードであるため、これ自身は識別子と紛らわしいです。しかし、Rust の構文ではキーワードでない識別子が 2 つ並ぶことはないため、曖昧性なく union を発見することができます。

4.2.15　トレイト

```
ItemTrait ::= ...
```

4.2.16　実装

```
ItemImpl ::= ...
```

4.2.17　型

```
Type ::= ...
```

4.2.18　式

```
Expression ::= ...
```

4.2.19　パターン

```
Pattern ::= ...
```

4.2.20　ブロック

```
Block ::= ...
```

4.2.21　文

```
Statement ::= ...
```

4.2.22　ジェネリックス

```
Generics ::= ...
```

4.2.23 where 節

```
WhereClauses ::= ...
```

4.2.24 マクロ呼び出し

```
ItemMacro ::= ...
```

4.2.25 パス

```
ModPath ::= ...
```

4.2.26 可視性

```
Visibility ::= ...
```

4.2.27 属性

```
OuterAttribute ::= ...

InnerAttribute ::= ...
```

第 5 章

構文拡張

第 6 章

名前解決

6.1　モジュール

6.2　インポート解決

6.3　レキシカルスコープの解決

6.4　型依存の名前解決

6.5　生存期間の解決

第 7 章

型検査

第 8 章

コード生成

8.1 MIR

8.2 LLVM IR

第 9 章

ドキュメンテーション

9.1　doc-comment の構文

参考文献

[1] Mark Davis. UAX #31: Unicode identifier and pattern syntax. Technical report, Unicode Inc., 2016. http://www.unicode.org/reports/tr31/tr31-25.html.

[2] Mark Davis, Laurențiu Iancu, and Ken Whistler. UAX #44: Unicode character database. Technical report, Unicode Inc., 2016. http://www.unicode.org/reports/tr44/tr44-18.html.