BogenLR Dokumentation

June 30, 2011

Contents

1 Module `BogenGrammar` : Datenstrukturen fuer die interne Darstellung von kontextfreien Grammatiken und einige Operationen darauf.

`exception Token_already_exists of string`

Ein Terminalsymbol darf innerhalb einer Grammatikbeschreibung nicht mehrfach deklariert sein.

`exception Startsymbol_already_exists`

Die Deklaration mehrerer Startsymbole ist nicht vorgesehen.

`exception Token_as_lhs of string`

Ein Terminalsymbol darf nicht auf der linken Seite einer Grammatikregel auftreten.

`exception No_rule_for_startsymbol of string`

Wenn ein Startsymbol explizit deklariert wird, muss es auch eine Regel dafuer geben.

`exception Unreachable_nonterminal`

Jedes Nichtterminal, das durch eine Regel deklariert ist, sollte auch erreichbar sein, d.h. auf der rechten Seite einer Produktion auftreten.

`exception Undefined_nonterminal`

Jedes in der Grammatik benutzte Nichtterminal muss durch eine Regel definiert werden.

```
type nonterminal_t = {
  number : int ;
```

Laufende Nummer des Symbols

```
  items : int list ;
```

Liste von Nummern derjenigen Items [A → . \alpha_j], die bei Expansion anhand A die Folgeknoten im Graphen werden.

```
  first_k : string list list ;
```

Die FIRST_k-Menge des Symbols, repraesentiert als Liste von Listen von Strings. Dabei enthaelt jeder String die Interdarstellung fuer ein Terminalsymbol.

```
}
```

Datentyp fuer die Eintraege in der Hashtabelle der Nichtterminalsymbole.

```
type terminal_t = int
```

Der Tabelleneintrag fuer ein Terminalsymbol besteht aus seiner Nummer in einer bei 258 beginnenden fortlaufenden Nummerierung.

```
type symbol_t =
  | Nonterminal of string
  | Terminal of string
  | Epsilon
```

Nichtterminal- und Terminalsymbole der Grammatik werden in der internen Darstellung fuer Grammatikregeln durch Strings repraesentiert, mit vorangestelltem Kontruktor `Nonterminal` oder `Terminal`, der den Symboltyp kennzeichnet. Das leere Wort wird in Items durch den Konstruktor `Epsilon` dargestellt.

```
type rule_t = int * string * symbol_t list
```

Eine einzelne Grammatikregel wird dargestellt als Tripel aus:

- Nummer der Regel
- linke Seite = Nichtterminalsymbol als String
- rechte Seite = Folge von Terminal- und Nichtterminalsymbolen Produktionen mit mehreren Alternativen werden intern als einzelne Regeln gespeichert, d.h. es kann mehrere Regeln mit identischer linker Seite geben.

```
type t = {
  nonterminals : (string, nonterminal_t) Hashtbl.t ;
  terminals : (string, terminal_t) Hashtbl.t ;
```

```
  productions : rule_t list ;
  startsymbol : string ;
  header_start : int ;
  header_end : int ;
  trailer_start : int ;
}
```

Ein Element vom Typ `BogenGrammar.t` enthaelt eine interne Repraesentation der Eingabegrammatik, bestehend aus:

- Menge der Nichtterminalsymbole als Hashtabelle,
- Menge der Terminalsymbole als Hashtabelle,
- Liste der Produktionen,
- Startsymbol als String,
- Positionen von Anfang und Ende der Headers und Anfang des Trailers in der Eingabedatei, jeweils als Integer.

`val string_of_symbol : symbol_t -> string`

Hilfsfunktion, um aus einem Grammatiksymbol vom Typ `symbol_t` einen String zu erzeugen.

`string_of_symbol s` liefert die Stringrepraesentation von `s`.

Returns Stringrepraesentation des Symbols

`val string_of_nt_tbl : (string, nonterminal_t) Hashtbl.t -> string -> string`

Hilfsfunktion zur einfachen textuellen Ausgabe der Hashtabelle aller Nichtterminalsymbole.

`string_of_nt_tbl tbl sep` fuegt fuer jeden Eintrag in `tbl` im Ergebnisstring einen Teilstring der Form "<Schluessel> = <Stringdarstellung des Datenfeldes>" an.

Returns String, der Stringrepraesentationen aller Schluessel- und der zugehoerigen Datenfelder der Hashtabelle enthaelt, getrennt durch `sep`.

`val string_of_token_tbl : (string, int) Hashtbl.t -> string -> string`

Hilfsfunktion zur einfachen textuellen Ausgabe der Hashtabelle aller Tokens.

`string_of_token_tbl tbl sep` verwirft fuer jeden Eintrag in `tbl` das Datenfeld und gibt nur den Schluessel aus.

Returns String, der alle Schluesselwerte der Hashtabelle enthaelt, getrennt durch `sep`.

`val string_of_rule : int * string * symbol_t list -> string`

Hilfsfunktion zur textuellen Ausgabe einer Grammatikregel.

`string_of_rule (num, lhs, rhs)` liefert einen String, der die uebergebene Regel repraesentiert.

Returns String der Form "<num>: <lhs> → <rhs als leerzeichen-separierter String>".

`val string_of_grammar : t -> string`

`string_of_grammar g` erzeugt eine einfache Stringrepraesentation der internen Grammatikrepraesentation `g`.

Returns Stringdarstellung der Grammatik.

`val exists_rule_for_symbol : t -> string -> bool`

`exists_rule_for_symbol g s` testet, ob die Grammatikspezifikation `g` eine Regel fuer das Nichtterminalsymbol enthaelt, das durch den String `s` repraesentiert wird.

Returns `true`, falls es in der Grammatik eine Regel fuer `s` gibt; sonst `false`.

`val all_nonterminals_reachable : t -> bool`

`all_nonterminals_reachable g` testet, ob alle in der Grammatikspezifikation `g` deklarierten Nichtterminale erreichbar sind, d.h. ob sie auch auf der rechten Seite von Regeln vorkommen. Eine Ausnahme stellt das Startsymbol der Grammatik dar, dieses muss nicht auf einer rechten Regelseite benutzt werden.

Returns `true`, falls alle Nichtterminale erreichbar sind; sonst `false`.

`val all_nonterminals_defined : t -> bool`

`all_nonterminals_defined g` testet, ob alle in der Grammatik `g` auftretenden Nichtterminale durch eine Regel definiert sind.

Returns `true`, falls alle Nichtterminalsymbole, die auf rechten Seiten von Regeln auftreten, durch Regeln definiert sind; sonst `false`.

`val test : t -> unit`

`test g` testet einige einfache Kriterien, um festzustellen, ob die Grammatik `g` eine gueltige Grammatikspezifikation sein kann:

- Gibt es eine Regel fuer das im Deklarationsteil festgelegte Startsymbol?
- Werden alle Nichtterminale erreicht, oder gibt es solche, fuer die zwar Regeln vorhanden sind, die aber auf keiner rechten Seite auftreten? Dies ist nur fuer das Startsymbol erlaubt.
- Sind alle Nichtterminale, die auf rechten Seiten von Regeln auftreten, durch eigene Regeln definiert?

Raises

- `No_rule_for_startsymbol` Im Deklarationsteil wurde ein Startsymbol festgelegt, aber es gibt keine Regel fuer dieses Symbol.
- `Unreachable_nonterminal` Es gibt ausser dem Startsymbol mindestens ein Nichtterminalsymbol, das durch eine Regel definiert wird, aber niemals auf der rechten Seite einer Regel auftritt.
- `Undefined_nonterminal` Es gibt mindestens ein Nichtterminalsymbol, das auf der rechten Seite einer Regel benutzt wird, das aber durch keine eigene Regel definiert wird.

2 Module `BogenLexer` : Lexikalische Analyse der Eingabegrammatik.

Automatisch generiert per `ocamllex` aus der Datei `bogenLexer.mll`.

Die verwendeten Tokentypen sind im Modul BogenParser definiert.

```
val __ocaml_lex_tables : Lexing.lex_tables
val token : Lexing.lexbuf -> BogenParser.token
val __ocaml_lex_token_rec : Lexing.lexbuf -> int -> BogenParser.token
val comments : int -> Lexing.lexbuf -> BogenParser.token
val __ocaml_lex_comments_rec :
  int -> Lexing.lexbuf -> int -> BogenParser.token
```

3 Module `BogenParser` : Syntaxanalyse der Eingabegrammatik.

Automatisch generiert via `ocamlyacc` aus der Spezifikationsdatei `bogenParser.mly`.

```
type token =
  | HEADER of (int * int)
  | DECL_TOKEN
  | DECL_START
  | DELIMITER of int
  | IDENTIFIER of string
  | COLON
  | PIPE
  | SEMICOLON
  | EOF
  | SEMANTIC_ACTION of string
```

Syntaxanalyse der Eingabegrammatik. Automatisch generiert via `ocamlyacc` aus der Spezifikationsdatei `bogenParser.mly`.

`val token_tbl : (string, BogenGrammar.terminal_t) Hashtbl.t`

Initialisierung einer Hashtabelle fuer die Terminalsymbole. Die Symbole werden in den Schluesselfeldern abgelegt, in den Datenfeldern wird jedem Symbol eine eindeutige Nummer zugewiesen.

`val nonterminal_tbl : (string, BogenGrammar.nonterminal_t) Hashtbl.t`

Initialisierung einer Hashtabelle fuer die Nichtterminalsymbole. Die Symbole werden in den Schluesselfeldern abgelegt. In den Datenfeldern werden bei der Generierung des Resultatparsers eine durchgehende Nummerierung sowie die FIRST_k-Mengen abgelegt.

`val startsym : string option Pervasives.ref`

Initialisierung des Startsymbols als leere Referenz.

`val token_number : BogenGrammar.terminal_t Pervasives.ref`

Die Tokens werden in der Reihenfolge ihrer Entdeckung fortlaufend durchnummeriert, beginnend bei 258.

`val nonterminal_number : int Pervasives.ref`

Die Nichtterminalsymbole werden in der Reihenfolge ihrer Entdeckung fortlaufend durchnummeriert, beginnend bei 1.

`val rule_number : int Pervasives.ref`

Die Regeln werden in der Reihenfolge ihrer Entdeckung fortlaufen durchnummeriert, beginnend bei 0.

```
val yytransl_const : int array
val yytransl_block : int array
val yylhs : string
val yylen : string
val yydefred : string
val yydgoto : string
val yysindex : string
val yyrindex : string
val yygindex : string
val yytablesize : int
val yytable : string
val yycheck : string
val yynames_const : string
val yynames_block : string
val yyact : (Parsing.parser_env -> Obj.t) array
val yytables : Parsing.parse_tables
val grammar : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> BogenGrammar.t
```

4 Module `BogenGenerator` : Stellt die Funktionen zur Verfuegung, die aus einer internen Grammatikrepraesentation vom Typ `BogenGrammar.t` einen Parser in Form von C-Programmcode erzeugen.

`val option_d : bool Pervasives.ref`

Flag, ob die Kommandozeilenoption "-d" beim Programmaufruf angegeben wurde. (Erzeugt Datei "<Ausgabedateiname>.h" mit Tokentypen fuer Lex.)

`val token_filename : string Pervasives.ref`

Stringrefernz fuer den Namen einer eventuellen Ausgabedatei fuer die Tokentypen, wie Lex sie benoetigt.

`val lookahead_length : int Pervasives.ref`

Laenge des Lookahead.

`val item_count : int Pervasives.ref`

Zaehler fuer die Anzahl der bisher insgesamt erzeugen Items bzw. fuer die hoechste schon verbrauchte Itemnummer. Die Werte 0 und 1 sind reserviert fuer die Items [S' → .S] und [S' → S.], die fuer Beginn und Ende des Parsevorganges stehen.

```
type item_t = {
  number : int ;
  dot_symbol : BogenGrammar.symbol_t ;
  dot_pos : int ;
  rule_number : int ;
  rule : string ;
  first_k : string list list ;
}
```

Datentyp fuer Items mit folgenden Feldern:

- `number`: Laufende Nummer, ueber die das Item angesprochen wird.
- `dot_symbol`: Welches Symbol steht direkt hinter dem Bearbeitungspunkt?
- `dot_pos` : Zu Diagnosezwecken Position des Bearbeitungspunktes.
- `rule`: Zu Diagnosezwecken Stringrepraesentation der Grammatikregel, auf der das Item basiert.
- `first_k`: FIRST_k-Menge der rechten Seite des Items, repraesentiert als Liste von Listen von Strings. Jede Liste steht dabei fuer ein Element der FIRST_k-Menge, jeder String in solch einer Liste steht fuer ein Terminalsymbol.

`val create_c_tokentypes : BogenGrammar.t -> string`

Die Funktion `create_c_tokentypes ig` erzeugt einen C-Aufzaehlungstyp fuer die Tokentypen in der Grammatik `ig`.

Returns String mit C-Code.

`val create_c_nonterminals : BogenGrammar.t -> string`

`create_c_nonterminals ig` erzeugt aus der Nichtterminal-Hashtabelle von `ig` C-Code, der die Nichtterminalmenge der Grammatik repraesentiert.

Returns Ein String aus C-Code fuer die Nichtterminalsymbole.

`val minlength : 'a list list -> int`

Der Funktionsaufruf `minlength l` berechnet fuer eine Liste `l` von Listen die Laenge der kuerzesten enthaltenen Liste.

Returns Laenge der kuerzesten Listen in l.

`val take : int -> 'a list -> 'a list`

`take n l` liefert exakt die ersten `n` Elemente der Liste `l`. Sollte die Liste zu wenig Elemente enthalten, werfen `hd` oder `tl` eine Exception.

Returns Liste mit den ersten n Elementen von `l`.

`val take_le : int -> 'a list -> 'a list`

`take_le n l` liefert die ersten `n` Elemente der Liste `l`. Sollte die Liste weniger Elemente enthalten, wird die gesamte Liste zurueckgegeben.

Returns Liste mit den ersten $\leq$ `n` Elementen von `l`.

`val take_uniq : int -> 'a list list -> 'a list list`

`take_uniq i l` nimmt eine Liste `l` von Liste und liefert eine neue Liste, in der alle Elementlisten aus bei Laenge i abgeschnitten sind. Das Resulat ist duplikatfrei.

Returns Liste mit den Elementlisten von `l`, abgeschnitten bei Laenge i und duplikatfrei.

`val remove : int -> 'a list -> 'a list`

`remove n l` entfernt die ersten `n` Elemente der Liste `l`. Sollte die Liste weniger als `n` Elemente enthalten, wird eine Exception geworfen.

Returns Liste `l` ohne die ersten `n` Elemente.

`val oplus : 'a list list -> 'a list list -> 'a list list`

Die Hilfsfunktion `oplus` vereinigt zwei Listen von Listen von Strings folgendermassen: `oplus l1 l2` konkateniert jede Subliste von l1 mit jeder Subliste von l2.

Returns Neue Liste, in der jede Liste in `l1` mit jeder Liste in `l2` konkateniert ist.

`val oplus_uniq : int -> 'a list list -> 'a list list -> 'a list list`

`oplus_uniq i l1 l2` liefert eine Liste `l`, in der alle Elementlisten aus `l1` mit allen Elementlisten aus `l2` konkateniert sind, wobei die Elementlisten von `l` bei einer Laenge von `i` abgeschnitten sind und `l` keine Duplikate enthaelt.

```
val compute_f0 :
  BogenGrammar.t ->
  (BogenGrammar.symbol_t, string list list * 'a list) Hashtbl.t ->
  string -> 'b -> unit
```

`compute_f0`: Hilfsfunktion zum Berechnen der F_0-Mengen fuer alle Nichtterminale im Initialisierungsteil von `compute_firstk_nt`: `compute_f0 ig f_table key data` fuegt zur Hashtabelle der F_0-Werte `f_table` fuer das Nichterminalsymbol `key` den Wert F_0(key) hinzu.

Returns `()`, da das Ergebnis direkt in die Hashtabelle `f_table` eingetragen wird.

```
val test_fi :
  ('a, 'b * 'c list list) Hashtbl.t ->
  'c list -> 'a list -> 'c list list -> bool * 'c list list
```

`test_fi` ist eine Hilfsfunktion fuer `compute_fi`: `test_fi f_table x rhs_rest f_so_far` ueberprueft, ob sich die bisher bestimmte Folge `x` von Terminalsymbolen weiter zu einem Eintrag in F_i(A) fuer ein Nichtterminalsymbol A fortsetzen laesst, indem nacheinander alle Symbole auf der rechten Seite der aktuell betrachteten Regel fuer A berueckichtigt werden.

Returns Paar aus changeflag und, ggf. ergaenzter, Liste von herleitbaren Terminal-Listen.

```
val compute_fi :
  BogenGrammar.t ->
  (BogenGrammar.symbol_t, 'a * 'b list list) Hashtbl.t ->
  string -> 'b list list -> bool * 'b list list
```

`compute_fi`: Hilfsfunktion zur Berechnung der F'_i-Mengen in `compute_firstk_nt`: `compute_fi ig f_table nt_key old_f` bildet fuer das Nichtterminalsymbol `nt_key` mittels der F_-Mengen in `f_table` aus der Menge F_(A), repraesentiert durch `old_f`, die neue Menge F_i(A), die den Rueckgabewert der Funktion darstellt.

Returns Paar aus changeflag und einer Liste, die F_i(A) repraesentiert.

```
val compute_firstk_nt : BogenGrammar.t -> unit
```

`compute_firstk_nt ig` dient der Vorausberechnung der FIRST_k-Mengen fuer alle Nichtterminalsymbole der Grammatik `ig` mittels Fixpunktiteration.

Returns `()`, da die Ergebnisse der Berechnung direkt in die Hashtabelle `ig.nonterminals` eingetragen werden.

```
val first_k_string :
  BogenGrammar.symbol_t list ->
  (string, BogenGrammar.nonterminal_t) Hashtbl.t -> string list list
```

Der Funktionsaufruf `first_k_string s nonterminals` bestimmt fuer die Symbolfolge, die durch die Liste `s` repraesentiert wird, die Menge FIRST_k(s).

Returns Liste von Listen aus Terminalsymbolen, wobei jede Subliste einen String aus FIRST_k(s) repraesentiert.

```
val items_of_rule' :
  (string, BogenGrammar.nonterminal_t) Hashtbl.t ->
  int * string * BogenGrammar.symbol_t list ->
  int -> item_t list
```

`items_of_rule' nonterminals (num, lhs, rhs) pos` erzeugt fuer die Regel `(num, lhs, rhs)` dasjenige Item, das den Punkt auf der rechten Seite an Position `pos` hat, sowie rekursiv alle Items mit einer frueheren Position pos', $0 <$ pos' $<$ pos des Punktes, bis 0. Die Teilergebnisse werden zu einer Resultatliste zusammengefasst. Dabei bedeutet `pos = 0`, dass sich der Punkt vor dem ersten Symbol befindet, und `pos = (List.length rhs)`, dass sich der Punkt hinter dem letzten Symbol befindet. Im Fall `pos = 0` wird ausserdem in der Nichtterminaltabelle `nonterminals` der Eintrag fuer das Symbol `lhs` ergaenzt, indem die Nummer des neu generierten Items dort vermerkt wird.

Returns Liste von Items vom Typ `item_t`.

```
val items_of_rule :
  (string, BogenGrammar.nonterminal_t) Hashtbl.t ->
  item_t list ->
  int * string * BogenGrammar.symbol_t list -> item_t list
```

`items_of_rule nonterminals items_so_far (num, lhs, rhs)` erzeugt eine Liste aller Items, die sich aus der aktuellen Regel `lhs, rhs` ergeben, indem die rekursive Hilfsfunktion `items_of_rule'` mit dieser Regel aufgerufen wird. Die Hilfsfunktion erzeugt alle Items, die sich durch die moeglichen Positionen des Bearbeitungspunktes von 0 bis `List.length rhs` ergeben koennen. Das Ergebnis dieses Aufrufs wird mit der bisherigen Ergebnisliste zusammengefuegt.

Returns Liste `items_so_far`, ergaenzt um alle Items, die aus der Regel `(num, lhs, rhs)` entstehen.

```
val items : BogenGrammar.t -> item_t list
```

`items ig` erzeugt eine Liste aller Items, die aus den Produktionen der Grammatik `ig` entstehen koennen. Besondere Items sind [S' → .S] und [S' → S.], die Beginn und Ende des Parse-Vorgangs markieren. Fuer sie sind die Nummern 0 und 1 reserviert.

Returns Liste von Items vom Typ `item_t`, bestehend aus:

- einer eindeutigen Nummer fuer das Item,
- dem Symbol direkt hinter dem Punkt (Typ `BogenGrammar.symbol_t`),
- Nummer und Stringrepraesentation der Regel, aus der das Item entstanden ist, und
- der FIRST_k-Menge fuer die rechte Seite des Items.

```
val create_c_items : BogenGrammar.t -> item_t list -> string
```

`create_c_items ig item_list` erzeugt aus einer Liste `item_list` von Items den entsprechenden C-Code als String.

Returns String mit C-Code.

```
val create_c_rules : BogenGrammar.t -> string
```

`create_c_rules ig` erzeugt C-Code, der die Regelmenge der Grammatik `ig` repraesentiert.

Returns String mit dem entsprechenden C-Code.

```
val create_headerfile : string -> unit
```

`create_headerfile token_string` erzeugt eine C-Quellcodedatei <token_filename> mit den Tokentyp-Deklarationen, die im String `token_string` als C-Code abgelegt sind. Benoetigt wird dies bspw., um dem Scannergenerator (f)lex die Tokentypen bekannt zu machen, die er an den Parser liefern soll.

Returns `()`

```
val copy_bogenfix : Pervasives.out_channel -> unit
```

`copy_bogenfix oc` kopiert den fixen Anteil des Parsers, der sich in der Datei "bogenfix.c" befindet, in die Ausgabedatei `oc`.

Returns `()`

```
val create_parser : BogenGrammar.t -> Pervasives.out_channel -> unit
```

`create_parser ig oc` erzeugt aus der internen Darstellung `ig` einer LR(k)-Grammatik den C-Code fuer den Resultatparser und schreibt ihn in den Ausgabekanal `oc`.

Returns `()`

5 Module `Bogenlr` : Hauptprogramm, das die Funktionen zum Einlesen der Eingabegrammatik und zur Erzeugung des C-Codes fuer den Parser aufruft.

Funktionsweise:

- Einlesen der Grammatikspezifikation aus einer Datei, deren Name dem Programm als Kommandozeilenparameter vorliegt.
- Einlesen der Grammatik und Erzeugen einer internen Repraesentation mittels der Module BogenLexer und BogenParser.
- Durchfuehrung von Tests aus dem Modul BogenGrammar auf der Grammatik, um sie auf einige notwendige Eigenschaften fuer LR(k)-Grammatiken zu ueberpruefen.
- Generierung des eigentlichen Parsers mit dem Modul BogenGenerator.
- Ausgabe des C-Code fuer den erzeugten Parser sowie von Header und Trailer aus der Eingabedatei in die Ausgabedatei.

Dabei greift das Programm `bogenlr` auf folgende Module zurueck:

- BogenGrammar: Stellt die Datenstrukturen fuer eine interne Repraesentation der Grammatik zur Verfuegung, zusammen mit grundlegenden Operationen darauf.
- BogenLexer: Scanner, der aus der Eingabedatei eine Folge von Token erzeugt und eine Fehlermeldung ausgibt, falls die Eingabe nicht lexikalisch korrekt ist. bogenLexer.ml wird mit `ocamllex` aus der Spezifikationsdatei bogenLexer.mll automatisch generiert.
- BogenParser: Ueberfuehrt die Eingabegrammatik, sofern sie syntaktisch korrekt ist, in eine interne Datenstruktur vom Typ `BogenGrammar.t`. bogenParser.ml wird mit `ocamlyacc` aus der Spezifikationsdatei bogenParser.mly automatisch generiert.

```
exception Wrong_number_of_inputfiles of int
```

Auf der Kommandozeile muss genau eine Eingabedatei spezifiziert werden.

```
val in_filename : string Pervasives.ref
```

Stringreferenz fuer den Namen der Eingabedatei.

`val out_filename : string Pervasives.ref`

Stringreferenz fuer den Namen der Ausgabedatei. Defaultwert: "bogenout.c"

`val option_r : bool Pervasives.ref`

Flag, ob die Option "-r" auf der Kommandozeile angegeben wurde:

`val cmdline_options : (Arg.key * Arg.spec * Arg.doc) list`

Definition der erlaubten Kommandozeilen-Optionen.

`val usage_msg : string`

Fehlermeldung bei ungueltiger Kommandozeile oder Aufruf mit Option "-help" oder "–help":

`val anon_count : int Pervasives.ref`

Zaehler fuer die Anzahl bisher gelesener anonymer Parameter auf der Kommandozeile.

`val anon_fun : string -> unit`

Die Funktion `anon_fun s` verarbeitet anonyme Kommandozeilenparameter, die also nicht an eine Option gebunden sind. Davon muss es bei einem korrekten Aufruf genau einen geben, naemlich den, der den Namen der Eingabedatei bestimmt.

Returns `()`, da im Erfolgsfall die Stringreferenz `in_filename` auf den Wert `s` festgelegt wird.

`val in_channel : string -> Pervasives.in_channel`

`in_channel x` stellt einen Eingabekanal fuer die Datei mit dem Namen `x` bereit. Wenn es keine Datei mit dem angegebenen Namen gibt oder sie aus anderen Gruenden nicht geoeffnet werden kann, beendet sich das Programm mit einer Fehlermeldung und dem Fehlercode 1.

Returns Eingabekanal.

`val out_channel : string -> Pervasives.out_channel`

`out_channel x` stellt einen Ausgabe-Kanal fuer die Datei mit dem Namen x bereit. Wenn die Datei bereits existiert, wird sie geloescht und als leere Datei neu erstellt. Wenn die Datei nicht geoeffnet werden kann, beendet sich das Programm mit einer Fehlermeldung und dem Fehlercode 1.

Returns Ausgabekanal.

`val in_grammar : Pervasives.in_channel -> BogenGrammar.t`

Die Funktion `in_grammar ic` ruft Scanner und Parser auf, um die Grammatikspezifikation aus dem Eingabekanal `ic` in eine interne Datenstruktur zu ueberfuehren. Sollte die Eingabedatei enden, bevor eine gueltige Grammatikspezifikation eingelesen wurde, beendet sich das Programm mit einer Fehlermeldung.

Returns Interne Grammatikdarstellung vom Typ `BogenGrammar.t`.

```
val output_header :
  Pervasives.in_channel -> Pervasives.out_channel -> int -> int -> unit
```

Die Funktion `output_header ic oc h_start h_end` kopiert den Headercode von Position `h_start` bis `h_end` unveraendert aus dem Quellkanel `ic` in den Zielkanal `oc`.

Returns `()`

```
val output_trailer :
  Pervasives.in_channel -> Pervasives.out_channel -> int -> unit
```

Die Funktion `output_trailer ic oc t_start` kopiert den Trailercode von Position `t_start` bis zum Dateiende unveraendert aus dem Quellkanal `ic` in den Zielkanal `oc`.

Returns `()`

```
val output_report : BogenGrammar.t -> unit
```

Ausgabe eines Reports mit Informationen zur Eingabegrammatik und zum erzeugten Parser:

Returns `()`

```
val main : unit -> unit
```

Die Hauptfunktion `main ()` ruft die Funktionen zum Einlesen und Ueberpruefen der Eingabegrammatik sowie zum Generieren und zur Ausgabe des Resultatparsers auf.