Grammar Tool (gt)

User's manual

Version 1

Josh Meyer and Aaron Stump
with contributions from Vilhelm Sjöberg

August 1, 2011

Contents

Part I

User Manual

Chapter 1

Introduction

OCamlyacc generates a parser from a set rules that are defined by the user in syntax similar to BNF, and OCamllex translates a set of regular expressions to a lexical analyzer. The goal of gt is to allow the user to easily define a context-free grammar in one file and then generate the corresponding parser, lexical analyzer and syntax trees based on the rules defined in the file. In addition, gt adds support for EBNF notation. Here is an example of an input file:

```
1  lists
2
3  Grammar : grammar -> { olist }*.
4
5  OList : olist -> LSBRACKET [ element { SEMI element }* ] RSBRACKET.
6
7  Element :  element -> VAR | INT.
8
9  LSBRACKET = "[".
10 RSBRACKET = "]".
11 SEMI = ";".
12
13 VAR = {{ ['a'-'z''A'-'Z'] ['a'-'z''A'-'Z''0'-'9''_']* }}.
14 INT = {{ ['0'-'9']+ }}.
```

Gt takes a file in the format shown above and generates multiple OCaml source files, as show in Figure 1.1. These files can then be compiled to an executable.

After the files in Figure 1.1 are compiled the executable is ready to parse programs such as, [0 ; a ; 1], to syntax trees like the one shown in Figure 1.2. The syntax tree in Figure 1.2 was actually generated, in Graph-viz format, by the same executable.

Along with the ability to easily generate parsers, gt generates many useful functions. For example dump_gviz, which creates a Graph-Viz file containing a syntax tree definition for a given parseable file. The file names listed below are

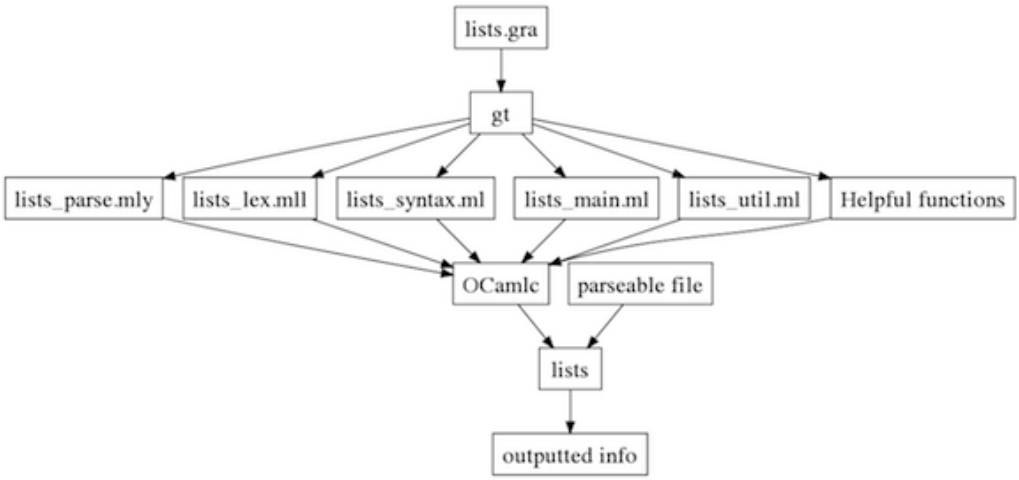

Figure 1.1

some of names corresponding to the gt generated files. Below the keyword *grammar_name* refers to the name of the grammar. For example, for the definition above *grammar_name* would would refer to lists *lists*.

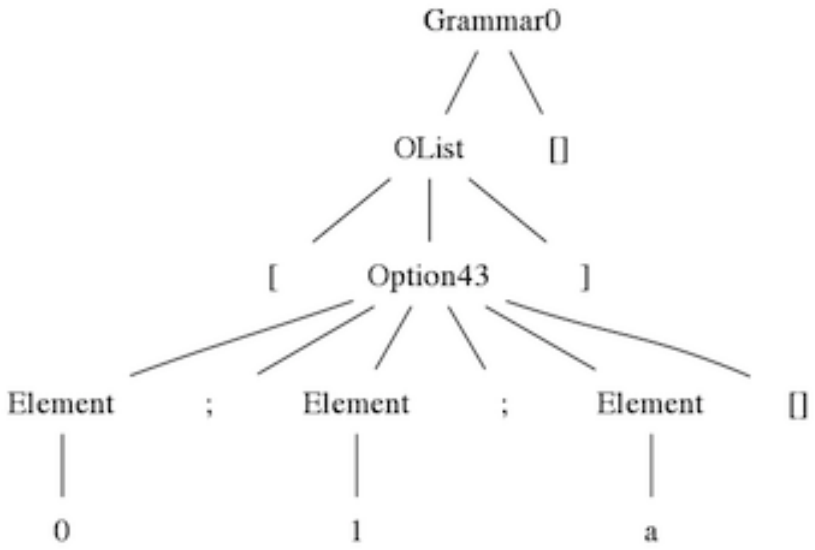

Figure 1.2

Generated Files	**Function Type**
grammar_name_gviz.ml	$(string \rightarrow unit) \rightarrow bool \rightarrow string \rightarrow start\_sym \rightarrow unit$
grammar_name_eq.ml	$(start\_sym * start\_sym) \rightarrow bool$
grammar_name_pp.ml	$(string \rightarrow unit) \rightarrow bool \rightarrow start\_sym \rightarrow unit$
grammar_name_ppast.ml	$(string \rightarrow unit) \rightarrow bool \rightarrow start\_sym \rightarrow unit$
grammar_name_syntax.ml	$start\_sym \rightarrow (int * string)$

Generated Files	**Function**
grammar_name_parse.mly	Generated Parser
grammar_name_lex.mll	Generated Lexical Analyser
grammar_name_main.ml	Main function
grammar_name_util.ml	Position data

1.1 Tutorial

In this section we will explain the basic format of a gt grammar definition. Later we will show complete examples. To look at specific grammar definitions please read Chapter 2, Examples. Alternatively, there are more examples in the *gt/tests* directory.

1.1.1 GT File Format

A grammar file has three basic parts: the name of the grammar, a line comment declaration and the body. The body of a grammar file is broken up into two sections: the productions and the lexical definitions.

The first part of a grammar definition is the grammar name. The grammar name will be used to prefix the names of the files generated by gt. It will also be the name of the default compiled executable, which parses string to syntax trees and can then print them back. The next part of the definition is where a user can define a line-comment. This is optional and if left out the default line-comment delimiter is the hash symbol, #. Lastly, the body of a grammar definition is where most of the work in defining a context-free grammar is done. The body is split into two sections, one for production definitions and the other for lexical class definitions.

```
1 Expr : factor -> RPAREN expr LPAREN.
2 Int : factor -> INT.
```

In the example above the first ID, *Expr*, is used by gt when defining OCaml constructors and therefore needs to be unique for each production and start with an uppercase letter. The next ID is used by gt for the name of the corresponding Ocaml types and should start with a lowercase letter. Next are the elements of the production. The elements of a production will define what a production

can shift or reduce to. Elements will be used by gt to define the corresponding OCaml types of the current production. The following is the OCaml code that would be generated by gt for the gt definition shown above. These types are defined and used for syntax trees.

```
(* pd holds position data and file name. *)
type __terminal__ = (pd * string);;
type __term_not_in_ast__ = pd;;

type factor =
    | Int of pd * __terminal__
    | Expr of pd * __term_not_in_ast__ * expr * __term_not_in_ast__
        ;;
```

In the OCaml code above ___*terminal*___ represents a lexical definition that is in the abstract syntax tree such as *{{['0'-'9']+}}*. Where as lexical definitions such as, *"+"* are represented by the ___terminal_not_in_ast___ type. The next section of the body is where the lexical classes are defined. These can be defined as either a string (eg. "+") or as a regular expression (e.g. {{ ['0'-'9']+ }}). The regular expression should be defined using OCaml regular expression syntax. Gt will store the lexical definitions that are in the abstract syntax tree with the string and position data of the corresponding lexical definition. Lexical definitions that are not in the abstract syntax tree only store the position data. Below is a simple example of a grammar definitions's lexical classes.

```
PLUS = "+".
INT = {{ ['0'-'9']+ }} .
```

The previous lexical definitions are compiled to the following OCaml lexical classes. These can be found in *grammar_name*_lex.mll.

```
...
| [ '0' - '9' ]+ as str { INT((StatSym_util.cur_pd(),str)) }
| "+" PLUS(StartSym_util.cur_pd())
...
```

1.2 Installing Grammar Tool (gt)

In this section there are a few basic steps to follow in order to install and use gt from the terminal. This guide is assuming that the user has already obtained gt. If you do not have it there is a list of links in Part II: Reference Manual where gt may be downloaded. First, make sure OCaml is installed on your computer. Gt is known to work with version 3.12.0 of OCaml, but it should work with older versions as it only uses the standard libraries. To test if your machine has OCaml open a terminal and type the following command.

```
ocaml -version
```

If OCaml your current machine has OCaml installed you should get a message saying the version of OCaml is found. If you do not have it go to the following page, download the latest version and follow the directions that are included in order to install OCaml.

http://caml.inria.fr/download.en.html

The following commands do *not* need to be executed unless files have been deleted or altered. First, navigate to *gt/* from the terminal. Next, execute the following commands:

```
make clean
make update
```

Lastly, if the user plans to view syntax trees then Graph-Viz is needed. To obtain Graph-Viz follow the link below to download the version that works on your machine. The user will still be able to generate the syntax trees without Graph-Viz, but they will not be able to view or export them without downloading and installing Graph-Viz.

http://graphviz.org/Download.php

A basic use of gt is shown as an example. There are other flags that are recognized by gt. To learn about these use the *./gt -help* command. Below *file* is referring to a path of a parseable grammar file. To generate a parser from the terminal, navigate to *gt/src/*. Then execute the following commands.

```
make clean
make
# for usage type ./gt -usage
./gt file
make -f grammar_name_Makefile
```

The command *make clean* will remove all of the compiled OCaml files. The *make* command will compile gt and generate an executable called gt, the grammar tool that will generate a parser from a grammar definition. To use the newly compiled executable use the command *./gt file* where *file* is a path to a parsable grammar definition. Now that gt has generated all of the necessary files, use the command *make emitted* or *make -f grammar_name_Makefile* to compile your parser. Once this has been done an executable with the same name as the *grammar_name* in the corresponding grammar definition has been created.

Now that executable is ready to be used, below is a walkthrough on how to use the newly generated parser. The following is how to use the compiled executable. This can be seen by typing *./grammar_name -help* in the terminal.

```
./grammar_name [options] <file>
    The options are:
       -p      Reprints the contents of <file> to
               <file>pp.txt. Without this option
               the contents are printed to the terminal.
       -a      Prints a text view of <file>'s AST to
               <file>ast.txt.
       -g      Outputs a Graphviz file named <file>gviz.dot
               that contains a visual representation of <file>'s
               syntax tree.
       -help   prints this
```

Chapter 2

Examples

This section has several examples of gt defined grammars. Each of the grammar definitions are explained in a step-by-step process. Alongside each of the walk-throughs are the parser and lexical definitions, generated functions and syntax trees.

2.1 BNF Notation

In the following section we will go through a step-by-step explanation of gt's BNF syntax. Later, this manual will explain gt's support for EBNF syntax.

2.1.1 Example - Simple

The following is a very basic grammar definition for gt. When completed the grammar will allow for one to many ID's where each is separated by a PLUS (eg. x + x). This example can be found in *gt/tests/bnf/simple.gra*.

First, a name for your grammar will need to be defined. A grammar name is the first declaration in a gt definition. It starts with a lowercase letter that is followed by zero or many upper- or lowercase letters, underscores and/or numbers (['a'-'z' 'A'-'Z'] ['0'-'9' '_' 'a'-'z' 'A'-'Z']*). We will name this grammar *simple*.

The next part of the grammar file that can be defined is the line_comment delimiter. This definition is optional. If the there is not a line_comment definition the default definition will be used – #.

```
1 simple
2
3 line_comment = "%".
```

The next step in creating a grammar definition is to define the productions. A production consists of a constructor name, type name and elements. The production name is a unique identifier and should start with an uppercase letter.

These are used by gt to define the corresponding OCaml constructors for the syntax trees parsed. A production also has a type name which should start with a lowercase letter. The type names do not have to be unique and are used by gt to name the corresponding OCaml types. Lastly, the elements of a production are a sequence of lexical, type names or nothing. These productions are used to declare OCaml types that will then be used to parse to syntax trees.

```
Expr : expr -> ID PLUS expr.
Id : expr -> ID.
```

The last step in creating a grammar is to define the lexical classes. The above example used ID and PLUS as some of the elements in the productions. To define the lexical classes the user must first decide what they want ID and PLUS to represent. To make it simple this example defined ID to be an x and PLUS to be a $+$. Putting all of this together we get the following grammar definition.

```
simple

line_comment = "%".

Expr : expr -> ID PLUS expr.
Id : expr -> ID.

PLUS = "+".
ID = "x".
```

Generated Functions

The files that are generated by gt for this grammar start with *simple*. For example, the equality function is called *simple_eq.ml* which is shown below. The equality function takes in a pair and returns true if the two elements in the pair are equal otherwise it returns false. The type for the equality function is $('a * 'a) \rightarrow bool$.

```
open Simple_syntax;;

let rec dummy () = ()
and eq_expr = function
  | Expr(d,pd1,pd2,expr3),Expr(d',pd1',pd2',expr3') ->
    true && eq_expr (expr3,expr3')
  | Id(d,pd1),Id(d',pd1') -> true
  | _ -> false;;

let eq e = eq_expr e;;
```

Another file that is outputted by gt is the file containing the pretty printer

function. As the name implies, pretty printer helps print the parsed program in a readable format. Instead of printing out the text that the generated parser has after creating the syntax trees, this function preserves the newlines of the original file. However, the pretty printer does not preserve comments, tabs or spaces. Below is the pretty printer definition for this example, *simple_pp.ml*.

```
let cur_line = ref 1;;
let rec print_new_line (os:string->unit)
   (do_print:bool) (p:int) : unit =
  if(p > !cur_line && do_print) then (
   os "\n";
   incr cur_line;
   print_new_line os do_print p;
  )
;;

let rec dummy () = ()
and pp_terminal (os:string->unit) (to_pretty_print:bool) = function
  | (d,str1) ->
   print_new_line os to_pretty_print (fst d);
   os str1
and pp_expr (os:string->unit) (to_pretty_print:bool) = function
  | Expr(d,pd1,pd2,expr3) ->
   print_new_line os to_pretty_print (fst d);
   print_new_line os to_pretty_print (fst pd1);
   os "x"; os " ";
   print_new_line os to_pretty_print (fst d);
   print_new_line os to_pretty_print (fst pd2);
   os "+"; os " ";
   pp_expr os to_pretty_print expr3; ()
  | Id(d,pd1) ->
   print_new_line os to_pretty_print (fst d);
   print_new_line os to_pretty_print (fst pd1);
   os "x"; os " "; ();;

let pp (os:string->unit) (to_pretty_print:bool) e =
  pp_expr os to_pretty_print e;;
```

The following is the output for the function that creates a Graph-Viz file. This function will create a visual representation of a syntax tree for a program that is parseable by the generated parser.

```
let cur_line = ref 1;;
let del = ref 0;;

let rec dummy () = ()
and gviz_terminal (os:string->unit)
   (to_pretty_print:bool) (cons:string) = function
 | (d,str1) ->
  print_new_line os to_pretty_print (fst d);
  let st = ref "" in
  let str1 =
    String.iter(fun s ->
```

```
12      if(s = "") then
13       st := ((!st)^"\\"^(Char.escaped s))
14      else
15       st := ((!st)^(Char.escaped s))
16     ) str1; !st
17   in
18   os str1
19 and gviz_expr (os:string->unit)
20    (to_pretty_print:bool) (cons:string) = function
21  | Expr(d,pd1,pd2,expr3) ->
22   let del' = !del in incr del;
23   let _ = del' in os "Expr";
24   os (string_of_int del'); os ";\n"; os"Expr";
25   os (string_of_int del'); os "[label=\""; os "Expr";
26   os "\"];\n"; os "Expr"; os (string_of_int del');
27   os " -- "; os "Expr"; os (string_of_int del');
28   os (string_of_int del'); os "_med0;\n";os "Expr";
29   os (string_of_int del'); os (string_of_int del');
30   os "_med0[label=\""; os "x"; os "\"];\n"; os "Expr";
31   os (string_of_int del'); os "[label=\""; os "Expr";
32   os "\"];\n"; os"Expr"; os (string_of_int del');
33   os " -- "; os "Expr"; os (string_of_int del');
34   os (string_of_int del'); os "_med1;\n"; os "Expr";
35   os (string_of_int del'); os (string_of_int del');
36   os "_med1[label=\""; os "+"; os "\"];\n";
37   os "Expr"; os (string_of_int del'); os "[label=\"";
38   os "Expr"; os "\"];\n"; os "Expr";
39   os (string_of_int del'); os " -- ";
40   gviz_expr os to_pretty_print (cons) expr3; ()
41  | Id(d,pd1) ->
42   let del' = !del in incr del;
43   let _ = del' in os "Id";
44   os (string_of_int del'); os ";\n"; os "Id";
45   os (string_of_int del'); os "[label=\""; os "Id";
46   os "\"];\n"; os "Id"; os (string_of_int del');
47   os " -- "; os "Id"; os (string_of_int del');
48   os (string_of_int del'); os "_med0;\n"; os "Id";
49   os (string_of_int del'); os (string_of_int del');
50   os "_med0[label=\""; os "x"; os "\"];\n"; ();;
51
52 let gviz (os:string->unit)
53    (to_pretty_print:bool) (cons:string) e =
54   os ("graph hopeful { \nnode"^
55      " [shape=\"plaintext\"];\n");
56   gviz_expr os to_pretty_print cons e; os "}";;
```

Syntax Trees

Now that the parser has been generated by gt it is ready to be used. A useful feature of gt is being able to generate visual syntax trees. The example will show the syntax tree that is generated by the above grammar. Figure 2.1 is an example of a that was generated by gt's gviz function.

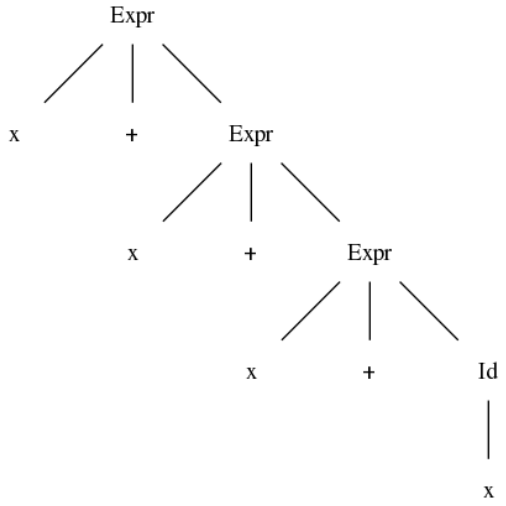

Figure 2.1: Syntax Tree for
$x + x + x + x$

2.2 EBNF Notation

Besides basic BNF support, gt also supports context-free grammars with EBNF notation. A user is able to define a production that has optional and repetitive types. The optional type allows an element(s) to have one or none and the repetitve feature allows element(s) to be zero or many, one or many or n or many where n is a non-negative integer. With the repetitive types a user can also define whether it is left or right associative. Gt's syntax for EBNF notation is explained below.

2.2.1 EBNF - Simple

The following example will go over the grammar definition in the file *ebnf-simple-exprs.gra* which can be found in *gt/tests/*. The goal of this grammar is to create parser that will successfully parse simple multiplication and addition expressions. An example of a parseable string would be:

0 + 09 * 23 + (48 * 33 + (88 + 23)) + 676

The name of this grammar is *expr*. The start symbol is also *expr* and is described as an *term* with zero or many terms separated by a *PLUS* after the initial *term*. Where the lexical class *PLUS* is '+' and a *term* is zero or many *factor TIMES* with a mandatory *factor*. The lexical class *TIMES* is defined as '*' and a *factor* can either be an *expr* surrounded by parenthesis or an *INT*.

Below is a walkthrough of this grammar.

Again the grammar name has been defined as *expr* and in this example a *line_comment* has not been defined and will be set to the default comment delimeter, #. Next, we need to define an *expr*. Below is a gt definition for the grammar explained earlier.

```
expr

Plus : expr -> term PLUS term.
```

Since multiplication is left associative this example uses the built in feature of gt to denote this. The following production name has been defined as *Mult*.

```
(* The following can also be written as:
Mult: term -> { factor TIMES }* factor *)
Mult : term -> { factor TIMES }(left,>=0) factor.
```

The last production we need to define is for factor. As explained earlier a factor can either be an can be an expr with parenthesis surrounding the expr or an INT.

```
Expr : factor -> LPAREN expr RPAREN.
Int : factor -> INT.
```

Next, we need to define the lexical classes for PLUS, TIMES, RPAREN, LPAREN and INT. PLUS and TIMES, RPAREN, LPAREN are lexical classes that are not in the abstract syntax tree. For INT we have to create a regular expression in OCaml syntax. Putting these steps together we get the final grammar definition which is can be found in the ebnf-simple-exprs.gra file.

```
expr

Plus : expr -> term PLUS term.

Mult : term -> { factor TIMES }(left,>=0) factor.

Expr : factor -> LPAREN expr RPAREN.
Int : factor -> INT.

PLUS = "+".
TIMES = "x".
LAPREN = "(" .
RPAREN = ")".
INT = {{ ['0'-'9']+ }}.
```

Generated Functions

Below is the generated parse file for the grammar defined above.

```
%{open Expr_syntax;;
  let parse_error s =
   let error =
    s^(Expr_util.string_of_pos
        (Expr_util.cur_pd()))
   in
   failwith error;; %}

%start main
%token EOF
%token <Expr_syntax.__term_not_in_ast__> LPAREN PLUS RPAREN TIMES
%token <Expr_syntax.__terminal__> INT
%type <Expr_syntax.expr option> main
%type <Expr_syntax.expr> expr
%type <Expr_syntax.factor> factor
%type <Expr_syntax.mult_term_factor4> mult_term_factor4
%type <Expr_syntax.plus_expr_plus0> plus_expr_plus0
%type <Expr_syntax.term> term
%type <Expr_util.pd> cur_position
%%

main:
  | expr { Some($1) }
  | EOF { None }

cur_position:
  | { Expr_util.cur_pd() }

expr:
  | term plus_expr_plus0 { Plus(pd_term $1, $1, $2) }

factor:
  | INT { Int(get_terminal_pd $1, $1) }
  | LPAREN expr RPAREN { Expr(get_term_pd_not_in_ast $1, $1, $2, $3
      ) }

term:
  | mult_term_factor4 factor { Mult(pd_mult_term_factor4 $1, $1, $2
      ) }

mult_term_factor4:
  | cur_position { ($1, []) }
  | mult_term_factor4 factor TIMES { (pd_mult_term_factor4 $1,
                          List.rev (($2, $3)::(List.rev (snd $1))))
                               }

plus_expr_plus0:
  | cur_position { ($1, []) }
  | PLUS term plus_expr_plus0 { (get_term_pd_not_in_ast $1, ($1, $2
      )::(snd $3)) }
```

Below is the generated lex file for the grammar defined above. More general descriptions of these files can be found in Part II of this manual.

```
{ open Expr_parse;; }

rule token = parse
  | ['\t' ' ' ]+ { token lexbuf }
  | '#'  (_ # ['\n' '\r'])* { token lexbuf }
  | ['\n' '\r']+ as str
    { Expr_util.line :=
      (!Expr_util.line + (String.length str));
      token lexbuf }
  |  ['0'-'9']+  as str { INT((Expr_util.cur_pd(),str)) }
  | "+" { PLUS(Expr_util.cur_pd()) }
  | "*" { TIMES(Expr_util.cur_pd()) }
  | "(" { LPAREN(Expr_util.cur_pd()) }
  | ")" { RPAREN(Expr_util.cur_pd()) }
  | eof { EOF }
  | _ {
   failwith((Lexing.lexeme lexbuf) ^ ": lexing error"^
           (Expr_util.string_of_pos
             (Expr_util.cur_pd())))}{}
```

Syntax Trees

Now that the parser for the given file has been generated by gt it is ready to be used. Figure 2.2 shows the syntax tree that is generated by the above grammar. Again, gt can dump the Graph-viz file, but without Graph-viz installed it is not possible to see the visual representation of a syntax tree or export them to images.

2.2.2 EBNF - Arith

This example is similar to the previous two, but we added a few more productions and lexical classes. We do this by adding unary operations and single length character variables to the grammar definiton. The most important differences in this example are the productions that utilize the optional and production ors notion and the lexical definition that uses the keyword *char*.

The first difference is the optional notation. Below we can see that *unary_op* is optional because it is between brackets and anything in between square brackets will be transformed into the OCaml optional type by gt when generating the parser.

```
Fact : factor -> [ unary_op ] element.
```

The next major change is the use of vertical bars to denote a type has multiple

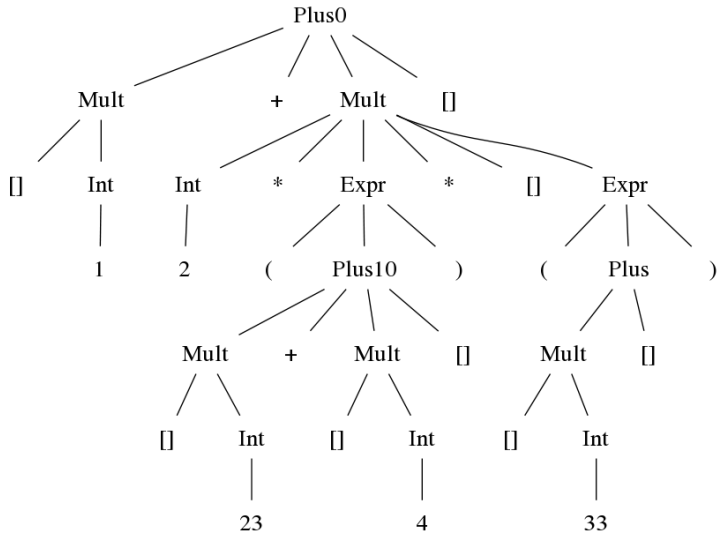

Figure 2.2: Syntax Tree for
*1 + 2 * (23 + 4) * (33)*

definitions. This can be very useful because it cuts back on defining extra productions. Instead of having three different productions for *element* we create one and utilize gt's EBNF notation to define the several different possibilities for element.

```
1 Element : element -> INT | VAR | LPAREN expr RPAREN.
```

The last major change is the use of the keyword *char*. If the regular expression for a lexical definition has a maximum length of one then this keyword is needed.

```
1 VAR = char {{ ['a'-'z''A'-'Z']}}.
```

Gt also supports basic definitions for Emacs and Gedit syntax modes. It is very simple to define basic syntax modes for the user created language using gt. The first thing the user needs to do is to define a file extensions. After a file extension is defined the user can map terminals that are not in the abstract syntax tree to predefined keywords. If the user does not define a syntax mode every terminal that is not in the abstract syntax tree will be mapped to *keyword* and the file extension will be the *grammar_name*.

```
1 file_ext = "gra"
```

```
keywords =
  ("x","+","-") : special
  (")","(") : keyword
```

Below is the whole grammar definition for the *arith* grammar. The file name is *ebnf-example.gra* and it can be located in the *gt/tests/ebnf* directory.

```
arith

Term : expr -> factor { op factor }(right,>=0).
Fact : factor -> [ unary_op ] element.
Element : element -> INT | VAR | LPAREN expr RPAREN.

Op : op -> PLUS | TIMES.
UnaryOp : unary_op -> MINUS.

PLUS = "+".
TIMES = "x".
MINUS = "-".
LAPREN = "(" .
RPAREN = ")".
INT = {{ ['0'-'9']+ }}.
VAR = char {{ ['a'-'z''A'-'Z']}}.

file_ext = "gra"
keywords =
  ("x","+","-") : special
  (")","(") : special
```

Generated Functions

Below is the generated syntax file for the grammar described.

```
open Arith_util;;

type __terminal__ = (pd * string);;
type __term_not_in_ast__ = pd;;

type dummy = Dummy
and element =
  | Element0 of pd * __terminal__
  | Element1 of pd * __terminal__
  | Element2 of pd * __term_not_in_ast__ * expr *
      __term_not_in_ast__
and expr = | Term of pd * factor * term_expr_op1
and factor = | Fact of pd * fact_factor_unary_op3 * element
and op =
  | Op0 of pd * __term_not_in_ast__
  | Op1 of pd * __term_not_in_ast__
and unary_op = | UnaryOp of pd * __term_not_in_ast__
and fact_factor_unary_op3 = pd * ( unary_op) option
and term_expr_op1 = pd * ( op * factor) list;;


```

```
let rec dummy () = ()
and get_terminal_pd = function
  | (pd,_) -> pd
and get_term_pd_not_in_ast = function
  | (pd) -> pd
and pd_element = function
  | Element0(pd,_) -> pd
  | Element1(pd,_) -> pd
  | Element2(pd,_,_,_) -> pd
and pd_expr = function
  | Term(pd,_,_) -> pd
and pd_factor = function
  | Fact(pd,_,_) -> pd
and pd_op = function
  | Op0(pd,_) -> pd
  | Op1(pd,_) -> pd
and pd_unary_op = function
  | UnaryOp(pd,_) -> pd
and pd_fact_factor_unary_op3 = function
  | (pd,Some(_)) -> pd
  | (pd,None) -> pd
and pd_term_expr_op1 = function
  | (pd,[]) -> pd
  | (pd,(_,_)::___tail___) -> pd;;

let pd e = pd_expr e;;
```

Below is the generated util file for the grammar described. A more general explanation of these files can be found in Part II of this manual.

```
let fname = ref
  (if(Array.length(Sys.argv) > 1) then
    (Sys.argv.(1))
  else ("<stdin>"));;

let line = ref 1;;
type pd = int * string;;

let string_of_pos (p:pd) =
  " on line "^(string_of_int (fst p))^" in file "^(snd p);;

let cur_pd () :pd = !line,!fname;;
```

Syntax Trees

Now that the parser for the given file has been generated by gt it is ready to be used. Figure 2.3 shows the syntax tree that is generated by the above grammar.

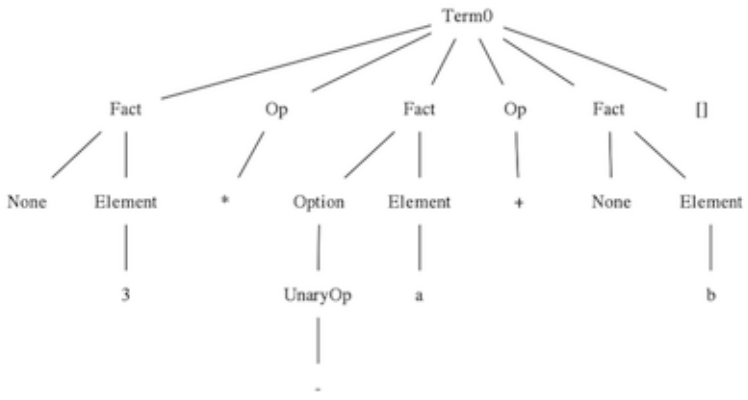

Figure 2.3: Syntax Tree for
*3 * -a + b*

Part II

Reference Manual

Chapter 3

Grammar Tool (gt)

3.1 Reference: GT Grammar

Below is a general description of gt's grammar. The actual grammar definition can be found in the parse.mly file located in *gt/src*. For examples of grammar files please read Chapter 2, Examples, or look at the .gra files in the gt/tests/ directory.

grammar	$\rightarrow^m$	*ID LINECOMMENT = IQUOTED . pls highlights*
grammar	$\rightarrow^m$	*ID pls highlights*
pls	$\rightarrow^m$	*production lexclasses*
pls	$\rightarrow^m$	*production pls*
production	$\rightarrow^m$	*ID : ID -> prod_body .*
production	$\rightarrow^m$	*ID : ID -> .*
prod_body	$\rightarrow^m$	*| elements prod_ors*
prod_body	$\rightarrow^m$	*elements prod_ors*
prod_ors	$\rightarrow^m$	*| elements prod_ors*
prod_ors	$\rightarrow^m$	
symbol	$\rightarrow^m$	*ID*
elements	$\rightarrow^m$	*element elements*
elements	$\rightarrow^m$	*element*

$$
\begin{array}{lll}
\mathit{element} & \rightarrow^m & \mathit{symbol} \\
\mathit{element} & \rightarrow^m & [\ \mathit{prod\_body}\ ] \\
\mathit{element} & \rightarrow^m & \{\ \mathit{prod\_body}\ \}\ (\ \mathit{lr}\ ,\ \mathit{GTE\ NONNEG}\ ) \\
\mathit{element} & \rightarrow^m & \{\ \mathit{prod\_body}\ \}\ + \\
\mathit{element} & \rightarrow^m & \{\ \mathit{prod\_body}\ \}\ * \\
 & & \\
\mathit{highlights} & \rightarrow^m & \mathit{EOF} \\
\mathit{highlights} & \rightarrow^m & \mathit{FILEEXT}\ =\ \mathit{IQUOTEDKEYWORDS}\ =\ \mathit{keywords} \\
 & & \\
\mathit{keywords} & \rightarrow^m & \mathit{keyword} \\
\mathit{keywords} & \rightarrow^m & \mathit{keyword\ keywords} \\
 & & \\
\mathit{keyword} & \rightarrow^m & (\ \mathit{IQUOTED\ qts}\ )\ :\ \mathit{tp} \\
 & & \\
\mathit{qts} & \rightarrow^m & \\
\mathit{qts} & \rightarrow^m & ,\ \mathit{IQUOTED\ qts} \\
 & & \\
\mathit{COMMENT} & = & \text{"comment"} \\
\mathit{NOTE} & = & \text{"note"} \\
\mathit{INTEGER} & = & \text{"integer"} \\
\mathit{FLOAT} & = & \text{"float"} \\
\mathit{DEC} & = & \text{"dec"} \\
\mathit{STRING} & = & \text{"string"} \\
\mathit{KEYW} & = & \text{"keyword"} \\
\mathit{TYPE} & = & \text{"type"} \\
\mathit{SPECHAR} & = & \text{"special"} \\
\mathit{BOOL} & = & \text{"bool"} \\
\mathit{ERR} & = & \text{"error"} \\
\mathit{META} & = & \text{"meta"} \\
 & & \\
\mathit{lr} & \rightarrow^m & \mathit{LEFT} \\
\mathit{lr} & \rightarrow^m & \mathit{RIGHT} \\
 & & \\
\mathit{lexclasses} & \rightarrow^m & \mathit{lexclass\ lexclasses} \\
\mathit{lexclasses} & \rightarrow^m & \mathit{lexclass} \\
 & & \\
\mathit{lexclass} & \rightarrow^m & \mathit{ID}\ =\ \mathit{quoted}\ . \\
 & & \\
\mathit{quoted} & \rightarrow^m & \mathit{QUOTED\ end\_quoted} \\
\mathit{quoted} & \rightarrow^m & \mathit{CHARQUOTED\ end\_quoted} \\
\mathit{quoted} & \rightarrow^m & \mathit{IQUOTED} \\
 & & \\
\mathit{end\_quoted} & \rightarrow^m & \mathit{END\_QUOTED}
\end{array}
$$

LEFT	=	”left”
RIGHT	=	”right”
GTE	=	”>=””
LINECOMMENT	=	”line_comment”
IQUOTED	=	a string enclosed in quotes with all ", \n and \ characters escaped
ID	=	a list of characters starting with a letter and followed by 0 or many letters, numbers and/or underscores
END_QUOTED	=	”}}”
QUOTED	=	anything except two consecutive ’}’ characters
CHAR	=	”char”
EOF	=	end of file
NONNEG	=	a non-negative integer

3.1.1 Gt Download Links

Below is a list of links to download gt.

- https://github.com/jjmeyer0/grammar-tool

3.1.2 Outputted Files

Gt outputs many files based on the grammar defined by the input file. Below we will explain the structure of the gt generated files. We will mostly show the the interfaces of the files and explain the major functions of each.

Util File: *grammar_name*_util.mli

The utility file contains functions that help gt keep track of the current line number and filename. There are also helper functions that allows gt to get the current position data and to print the line number and filename more easily.

```
1  (** The current file name of the file being parsed. *)
2  val fname : string ref
3
4  (** Keeps track of the current line for error reporting when
       parsing a file.*)
5  val line : int ref
6
7  (** The type declaration for position data. In this pair int
       represents the
8     position of the information and the string represents the file
          name of
9     the file being parsed.  *)
10 type pd = int * string
11
12 (** A function that takes a parameter of type pd and returns a
       string
```

```
13      that says
14
15      @param pd
16      @return str a string that says " on line <fst pd> in file <snd
            pd>" *)
17  val string_of_pos : pd -> string
18
19  (** This functions returns the current positions data when parsing
        a file.
20
21      @param unit
22      @return pd the current position data. *)
23  val cur_pd : unit -> pd
```

Syntax File: *grammar_name*_syntax.ml

The generated syntax file defines types that are based on the given grammar file. These types use the OCaml optional type when the corresponding grammar file uses the gt's optional feature and OCaml list type when the grammar definition uses the repetition feature of gt. Also, the The terminals that are not in the abstract syntax tree only contain the position data information where as the terminals in the abstract syntax tree contain the position data and string refering the terminal.

```
1   (** A type used for terminals that are in the ast. *)
2   type __terminal__ = Grammar_name_util.pd * string
3
4   (** A type used for terminals that are not in the ast. *)
5   type __term_not_in_ast__ = Grammar_name_util.pd
6
7   (** Just a dummy type to make it easier to generate types. *)
8   type dummy = Dummy
9
10  (** The type definition for element0. This type is used when
        building
11      syntax trees. *)
12  and element0 = Production0 of Grammar_name_util.pd * el0 * ... *
        eln | ... | Productionn of Grammar_name_util.pd * el0 * ... *
        eln
13
14    ...
15
16  (** The type definition for elementn. This type is used when
        building
17      syntax trees. *)
18  and elementn = Production0 of Grammar_name_util.pd * el0 * ... *
        eln | ... | Productionn of Grammar_name_util.pd * el0 * ... *
        eln
19
20
21  (** Just a dummy function to make it easier to generate functions.
        *)
22  val dummy : unit -> unit
```

```
23
24  (** A function to get the position data of a terminal that is in
        the ast.
25
26     @param terminal a terminal to get the position data.
27     @return 'a the position data of a terminal in the ast. *)
28  val get_terminal_pd : 'a * 'b -> 'a
29
30  (** A function to get the position data of a terminal that is not
        in the ast.
31
32     @param terminal a terminal to get position data from
33     @return 'a position data of a terminal not in an ast. *)
34  val get_term_pd_not_in_ast : 'a -> 'a
35
36  (** A function to get the position data of an element with type
        element0.
37
38     @param el find the position data of element0.
39     @return pd the positions data for el. *)
40  val pd_element0 : element0 -> Grammar_name_util.pd
41
42    ...
43
44  (** A function to get the position data of an element with type
        element0.
45
46     @param el find the position data of element0.
47     @return pd the positions data for el. *)
48  val pd_elementn : elementn -> Grammar_name_util.pd
49
50  (** A wrapper function that gets the position data of element0.
51
52     @param el find the position data of element0.
53     @return pd the positions data for el. *)
54  val pd : element0 -> Grammar_name_util.pd
```

Parse File: *grammar_name*_parse.ml

The top of the parser file sets the lexical classes to their corresponding type, whether it be *___term_not_in_ast___* or *___terminal___*. Then the productions are set to their corresponding OCaml types. The parse file that is generated by gt always starts with the *main* production which is of the OCaml type option. It will always be defined as being the start symbol of the grammar or end of file (eof). End of file is used in case the file inputed to the generated parser is empty. Each parser file also has a *cur_position* which returns the current position data at any given point when parsing. Gt transforms the gt grammar from EBNF to BNF to easily define the given grammar in OCamllex and OCamlyacc.

```
1  (** The type definitions for ther terminals of the defined language
       . *)
2  type token =
3    | EOF
```

```
  | TERMINAL_NOT_IN_AST0 of (Lists_syntax.__term_not_in_ast__)

    ...

  | TERMINAL_NOT_IN_ASTN of (Lists_syntax.__term_not_in_ast__)
  | TERMINAL_IN_AST0 of (Lists_syntax.__terminal__)

    ...

  | TERMINAL_IN_ASTN of (Lists_syntax.__terminal__)

val main : (Lexing.lexbuf -> token) -> Lexing.lexbuf ->
    Lists_syntax.grammar option
```

Lexical Classes File: *grammar_name*_lex.ml

The generated lexical file contains the OCamllex definitions for the gt defined lexical classes – the terminals of the gt defined context-free grammar.

```
{open Grammar_name_parse;;}

rule token = parse
  | ['\t' ' ' ]+ { token lexbuf }
  | '#'  (_ # ['\n' '\r'])* { token lexbuf }
  | ['\n' '\r']+ as str { Grammar_name_util.line := (!
      Grammar_name_util.line + (String.length str)); token lexbuf }
  | string_lit0 { NONASTTERMINAL0(Grammar_name_util.cur_pd()) }
  ...
  | string_litn { NONASTTERMINALN(Grammar_name_util.cur_pd()) }
  | regexp0 as str { ASTTERMINAL0((Grammar_name_util.cur_pd(),str))
      }
  ...
  |  regexn as str { ASTTERMINALN(Grammar_name_util.cur_pd(),str) }

  | eof { EOF }
  | _ {failwith((Lexing.lexeme lexbuf)^": lexing error"^(
      Grammar_name_util.string_of_pos (Grammar_name_util.cur_pd()))
      )}{}
```

Graph-Viz File: *grammar_name*_gviz.ml

This function allows the user to generate a visual representation of the syntax tree from a parsable program for the gt generated parser. The generated visual syntax tree can be viewed using Graph-Viz which is available for free on their website (http://www.graphviz.org/Download.php).

```
(** A reference to keep track of the current line number. *)
val cur_line : int ref

(** Used to create unique names when generating a gviz file. *)
val del : int ref

(** This function will print the right number of newlines in the
    correct
```

```
8     places for any parseable file.
9
10    @param os where to print the information
11    @param do_print if true newlines will be printed otherwise
          nothing will happen.
12    @param p if do_print is true then the difference between p and
          cur_line newlines will be printed.
13    @return unit *)
14  val print_new_line : (string -> unit) -> bool -> int -> unit
15
16  (** Just a dummy function to make it easier to generate functions.
      *)
17  val dummy : unit -> unit
18
19  (** Prints the given terminal.
20
21    @param os where to print the gviz information.
22    @param do_print if true then print newlines otherwise do nothing
          .
23    @param terminal the terminal to be printed.
24    @return unit *)
25  val gviz_terminal : (string -> unit) -> bool -> Lists_syntax.
      __terminal__ -> unit
26
27  (** A function that prints the correct gviz code for the element0
      type of
28    the given parseable file.
29
30    @param os where to print the information
31    @param do_print if true print newlines otherwise do nothing
32    @param cons a string of representing the previous type name.
33    @param x of type element0. This will be used to print the gviz
          information for element0.
34    @return unit *)
35  val gviz_element0 : (string -> unit) -> bool -> string ->
      Grammar_name_syntax.elemenet0 -> unit
36
37    ...
38
39  (** A function that prints the correct gviz code for the elementn
      type of
40    the given parseable file.
41
42    @param os where to print the information
43    @param do_print if true print newlines otherwise do nothing
44    @param cons a string of representing the previous type name.
45    @param x of type elementn. This will be used to print the gviz
          information for elementn.
46    @return unit *)
47  val gviz_elementn : (string -> unit) -> bool -> string ->
      Grammar_name_syntax.elementn -> unit
48
49  (** A wrapper function that will call the gviz function for the
      start
50    of the grammar.
51
52    @param os where to print the gviz information.
```

```
53      @param do_print initially false.
54      @param cons it is initially empty.
55      @param x the start symbol of the grammar.
56      @return unit *)
57  val gviz : (string -> unit) -> bool -> string ->
        Grammar_name_syntax.element0 -> unit
```

Pretty Printer AST File: *grammar_name*_ppast.ml

Pretty printer AST allows gt to print a parsed input file's syntax trees. These functions allow the user to print a textual representation of the syntax tree for the parsed file.

```
1   (** A reference to keep track of the current line number. *)
2   val cur_line : int ref
3
4   (** This function will print the right number of newlines in the
        correct
5       places for any parseable file.
6
7       @param os where to print the information
8       @param do_print if true newlines will be printed otherwise
            nothing will happen.
9       @param p if do_print is true then the difference between p and
            cur_line newlines will be printed.
10      @return unit *)
11  val print_new_line : (string -> unit) -> bool -> int -> unit
12
13  (** Just a dummy function to make it easier to generate functions.
        *)
14  val dummy : unit -> unit
15
16  (** Prints the given terminal.
17
18      @param os where to print the gviz information.
19      @param do_print if true then print newlines otherwise do nothing
            .
20      @param terminal the terminal to be printed.
21      @return unit *)
22  val ppast_terminal : (string -> unit) -> bool ->
        Grammar_name_syntax.__terminal__ -> unit
23
24  (** A function that prints a textual representation of a syntax
        tree.
25
26      @param os where to print the information
27      @param do_print if true print newlines otherwise do nothing
28      @param cons a string of representing the previous type name.
29      @param x of type elementn. This will be used to print the gviz
            information for elementn.
30      @return unit *)
31  val ppast_element0 : (string -> unit) -> bool ->
        Grammar_name_syntax.element0 -> unit
32
```

```
  ...

(** A function that prints a textual representation of a syntax
    tree.

   @param os where to print the information
   @param do_print if true print newlines otherwise do nothing
   @param cons a string of representing the previous type name.
   @param x of type elementn. This will be used to print the gviz
       information for elementn.
   @return unit *)
val ppast_elementn : (string -> unit) -> bool ->
    Grammar_name_syntax.elementn -> unit

(** A wrapper function that will call the gviz function for the
    start
   of the grammar.

   @param os where to print the gviz information.
   @param do_print initially false.
   @param cons it is initially empty.
   @param x the start symbol of the grammar.
   @return unit *)
val ppast : (string -> unit) -> bool -> Grammar_name_syntax.grammar
     -> unit
```

Equality File: *grammar_name*_eq.ml

The equality function takes in a pair and returns true if the two elements in the pair are equal, otherwise it returns false.

```
(** A reference to keep track of the current line number. *)
val cur_line : int ref

(** Just a dummy function to make it easier to generate functions.
    *)
val dummy : unit -> unit

(** A function to compare to parameters of type start_sym.

   @param els a pair of 2 elements to check if they are equal.
   @return b true if the are equal otherwise false *)
val eq_element0 : Grammar_name_syntax.start_sym *
    Grammar_name_syntax.start_sym -> bool

  ...

(** A function to compare to parameters of type elementn.

   @param els a pair of 2 elements to check if they are equal.
   @return b true if the are equal otherwise false *)
val eq_elementn : Grammar_name_syntax.elementn *
    Grammar_name_syntax.elementn -> bool

(** A wrapper function that will call the eq function for the start
```

```
22      of the grammar.
23
24      @param els a pair of 2 elements to check if they are equal.
25      @return b true if the are equal otherwise false *)
26  val eq : Grammar_name_syntax.start_sym * Grammar_name_syntax.
        start_sym -> bool
```

Pretty Printer File: *grammar_name*_pp.ml

Pretty printer allows gt to print a parsed input file in a readable format. This function prints a newline wherever there was one in the original file, but does not print out the comments of the input file.

```
1   (** A reference to keep track of the current line number. *)
2   val cur_line : int ref
3
4   (** This function will print the right number of newlines in the
        correct
5      places for any parseable file.
6
7      @param os where to print the information
8      @param do_print if true newlines will be printed otherwise
           nothing will happen.
9      @param p if do_print is true then the difference between p and
           cur_line newlines will be printed.
10     @return unit *)
11  val print_new_line : (string -> unit) -> bool -> int -> unit
12
13  (** Just a dummy function to make it easier to generate functions.
        *)
14  val dummy : unit -> unit
15
16  (** A function to pretty print for element0. This function will
        print newlines in
17     the correct location for element0.
18
19     @param os where to print the information
20     @param do_print if true print newlines otherwise do nothing.
21     @param el The start symbol of the grammar
22     @return unit *)
23  val pp_element0 : (string -> unit) -> bool -> Grammar_name_syntax.
        element0 -> unit
24
25    ...
26
27  (** A function to pretty print for elementn. This function will
        print newlines in
28     the correct location for elementn.
29
30     @param os where to print the information
31     @param do_print if true print newlines otherwise do nothing.
32     @param el The start symbol of the grammar
33     @return unit*)
34  val pp_elementn : (string -> unit) -> bool -> Grammar_name_syntax.
        elementn -> unit
```

```
35
36 (** The wrapper function of pp that will call the function for the
       start symbol.
37
38    @param os where to print the information
39    @param do_print if true print newlines otherwise do nothing.
40    @param el The start symbol of the grammar
41    @return unit *)
42 val pp : (string -> unit) -> bool -> Grammar_name_syntax.element0
       -> unit
```

Chapter 4

Meta Language for Common Functions (mlcf)

4.1 Reference: Generating Functions with MLCF

Below is a general description of mlcf's (meta language for common functions) grammar. The actual grammar definition can be found in the mlcf_parse.mly file in the *src* directory of mlcf or in the mlcf.gra file in the *mlcf* directory. It is not required to understand mlcf. A user will never have to use this tool unless one of the files that is created by it was deleted or broken. However, if a user would like to extend gt, mlcf may be usefule. Mlcf is basically a template for iterating through the productions of a gt grammar. It also, has common predefined functions that can be used when defining an mlcf function For examples of mlcf function please look at the mlcf in the *mlcf/functions/* directory. The mlcf functions modify *dump_mlcf_** files in *gt/src/*. The following grammar definition is based on mlcf.gra:

program	$\rightarrow^m$	[*ID ENDEACH = IQUOTED*]^m [*MLGLOBAL*]^m *const* { *body* }$*^m$
const	$\rightarrow^m$	*ID (ID,ID ... ,ID) ->*
body	$\rightarrow^m$	*OCAMLGLOBAL*
body	$\rightarrow^m$	*METACODE ID SEPBY*

ENDEACH	=	"end_of_each_match"
MLGLOBAL	=	starts with {{, ends with }} and allows anything in between
METACODE	=	starts with (~, ends with ~| and allows anything in between
SEPBY	=	starts with ~|, ends with ~) and allows anything in between
ID	=	A string that starts with an upper/lower-case letter followed by numbers, letters and/or _
IQUOTED	=	standard string literal

4.1.1 Example: Pretty Printer function

There are four basic parts to an mlcf definition: the function name, a code block before iterating through the productions, the body of the iteration function and a code block that is to be executed after iterating through the productions. The function name – below it is defined as *pp* – is used in the filename of the dumped function and in naming some predefined functions that are used when defining mlcf functions.

The code blocks can have any valid OCaml code in them. The iteration block – where most of the OCaml code is written – has three parts: the code block, the counter variable and with what to separate each iteration. There are a few predefined functions is_list, is_opt, is_terminal, is_in_ast and string_of_terminal that can be used in any mlcf program definition. Below the function pp_i refers to the pretty printer function name that will be used for the current element. This function will always be named *function_name*_i. The function x_i refers to the element to pretty print and it will be named with the first letter of *x1* followed by *_i*.

Below is the complete mlcf definition for the pretty printer function.

```
pp

PrettyPrinter (P, x1,...,xn)->
{{ emit_pattern 0; os " -> "; }}

(~
if(is_terminal) then (
  os ("print_new_line os to_pretty_print (fst d); ");
  if(is_in_ast) then (
    os " pp_terminal os to_pretty_print "
  ) else (
    os ("print_new_line os to_pretty_print (fst "^x_i^"); ");
    os "os "; os string_of_terminal
  );
) else (
  os pp_i; os " os to_pretty_print "
);

if(is_cur_eq_first_nt && (is_list || is_opt)) then (
  os ( " (d,"^x_i^")" );
) else if not(is_terminal && not is_in_ast) then (
  os ( x_i );
);

if(is_terminal) then os "; os \" \" ";

~| i ~| ; ~)

{{ os " () "; }}
```