gff: A Tool for Computing FIRST and FOLLOW Sets

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721.

1 Description

gff is a simple tool that takes as input a contex free grammar, and produces as output the FIRST and FOLLOW sets for the nonterminals of the grammar. It's a pretty bare-bones no frills tool, with only rudimentary error checking and handling.

As a simple example, consider the CFG

$$S \longrightarrow (S) \mid \varepsilon$$

To compute the FIRST and FOLLOW sets for this grammar, we first create a file—say, `ex.in`—specifying the grammar:

`ex.in:`

```
%%
S : '(' S ')'
  | /* epsilon */
  ;
```

The first line of this file, '%%', is a delimiter that separates some optional declarations (whicch come before the %% delimiter—in this case, there are no declarations) from the grammar productions. The syntax of the input files is discussed below.[1]

gff is invoked on this file as follows (with '%' being the shell prompt):

```
% gff ex.in
```

and produces the following output:

```
FIRST sets:
  S:   '(' <epsilon>

FOLLOW sets:
  S:   <EOF> ')'
```

[1] People acquainted with the tools `yacc` or `bison` will see strong similarities between the syntax of gff input fi les and those for `yacc`/`bison`. This is not an accident.

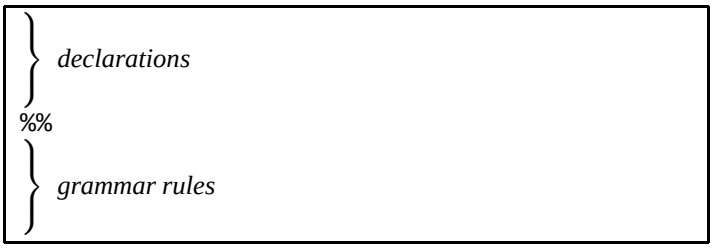
2 gff Usage

gff takes a single argument, the name of a file specifying a context free grammar. It produces as output, on `stdout`, the FIRST and FOLLOW sets of the nonterminals of that grammar.

3 Input Syntax

3.1 File Structure

An input file consists of two parts: a *declarations* part and a *productions* part. The two parts are separated by the delimiter '%%' on a line by itself. The file therefore has the following structure:



Both the declarations and the grammar rules are optional, but the delimiter between them si required. Thus, the shortest legal gff input would be the file consisting simply of the '%%' delimiter.

3.2 Comments

Comments are as in C: a character sequence (possibly spanning multiple lines) starting with '`/*`' and terminated by '`*/`', with no occurrences of '`*/`' in between.

Comments can appear anywhere in the input file except for the line containing the '%%' delimiter.

3.3 Identifiers

Identifiers are used to name nonterminals and, possibly, terminals. An identifier is as in C, and consists of a letter followed by zero or more letters, digits, and underscores.

Examples: `ID`, `IntegerConst`, `expression`, `stmt_list`

3.4 Declarations

The optional declarations section specifies identifiers that are token names, as well as the start symbol of the grammar.

3.4.1 Tokens

Tokens are declared using the '`%token`' keyword. Tokens that are given names, e.g., "IDENT" (identifier) or "INTCONST" (integer constant), must be declared as tokens; "character constant" tokens (a single character preceded and followed by a single quote) e.g., `'+'`, `'('`, need not be explicitly declared.

The declarations section can contain zero or more token declarations. Each such declaration is of the form

`%token` *id_list*

where *id_list* is a list of token names separated by whitespace. Each token name is an identifier. As an example, suppose that the identifiers we want to declare are: ID, INTCONST, STRINGCONST, and CHARCONST. The following are all acceptable ways to do this:

1. A single '`%token`' declaration with a list of identifiers:

```
%token ID INTCONST STRINGCONST CHARCONST
```

2. Multiple '`%token`' declarations, each with a list of identifiers:

```
%token ID INTCONST
%token STRINGCONST CHARCONST
```

3. Multiple '`%token`' declarations, each with a single identifier:

```
%token ID
%token INTCONST
%token STRINGCONST
%token CHARCONST
```

3.4.2 The Start Symbol

The start symbol of the grammar (which affects FOLLOW sets) can be specified explicitly, in the declarations portion of the input file, using the '`%start`' keyword:

`%start` *id*

where *id* is an identifier that is the name of a nonterminal.

Start symbol specifications are optional: if no start symbol is specified, the first nonterminal listed in the *grammar rules* section of the input file will be taken to be the start symbol. A grammar should not specify more than one start symbol: if multiple '`%start`' declarations are encountered in an input file, the first one is used.

As an example, suppose we have a grammar with nonterminals X and Y, where Y is the start symbol. The following would explicitly declare Y to be the start symbol:

```
%token ...
%start Y
%%
... grammar rules for X ...
... grammar rules for Y ...
```

The start symbol could also be declared implicitly, as follows, where Y is taken as the start symbol because it is the first nonterminal whose rules are listed in the grammar rules portion of the input:

```
%token ...
%%
... grammar rules for Y ...
... grammar rules for X ...
```

3.5 Grammar Rules

The grammar rules section of the input file consists of a list of nonterminals and their productions;

```
...
%%
nonterminal_1's productions
...
nonterminal_n's productions
```

3.5.1 Nonterminals

Nonterminal names are identifiers (see Section 3.3). Any identifier that has not been declared to be a token (see Section 3.4.1) is taken to be a nonterminal.

3.5.2 Productions

The productions for a nonterminal are specified as follows. Given a nonterminal X with productions

$$X \rightarrow \alpha_1$$
$$X \rightarrow \alpha_2$$
$$\cdots$$
$$X \rightarrow \alpha_n$$

the productions would be written as: the nonterminal X; then a colon delimiter; then a list of the right hand sides of its productions separated by '|'; and finally ending with a semicolon:

$$X : \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n ;$$

There is exactly one such specification for each nonterminal in the grammar. Thus, for a grammar with nonterminals $X_1, \ldots, X_k$, the grammar rules $X_1, \ldots, X_k$, where nonterminal X_i has productions

$$X_i \rightarrow \alpha_{i1} \mid \cdots \alpha_{in_i}$$

the productions would be specified as

```
...
%%
X_1 :   α_11 | ··· | α_1n_1 ;
...
X_k :   α_k1 | ··· | α_kn_k ;
```

If one of the right hand sides is ε, it is written simply as expected, e.g.: the productions

$$S \rightarrow (S) \mid \varepsilon$$

would be written as:

```
...
%%
S : '(' S ')'
  | /* epsilon */
;
```

4 An Example

Consider the context free grammar $G = (V, T, P, E)$, where:

- $V = \{E, E_1, T, T_1, F\}$ is the set of nonterminals;
- $T = \{\textbf{id}, \textbf{intcon}, +, -, *, /, (,)\}$ is the set of terminals;
- the start symbol is E; and
- the set of productions P consists of the following:

$$
\begin{array}{l}
E \rightarrow T\ E_1 \\
E_1 \rightarrow +\ T\ E_1 \mid -\ T\ E_1 \mid \varepsilon \\
T \rightarrow F\ T_1 \\
T_1 \rightarrow *\ F\ T_1 \mid /\ F\ T_1 \mid \varepsilon \\
F \rightarrow \textbf{id} \mid \textbf{intcon} \mid (\ E\ )
\end{array}
$$

A gff input file specifying this grammar would be:

```
%token ID INTCON
%start E
%%
E : T E1
;

E1 : '+' T E1
   | '-' T E1
   |  /* epsilon */
;

T : F T1
;

T1 : '*' F T1
   | '/' F T1
   |  /* epsilon */
;

F : ID
  | INTCON
  | '(' E ')'
;
```

Some style comments

It simplifies life and improves readability (especially if you're going to be making changes to the grammar) to have each right-hand-side of a production listed on a separate line, as in the example above. Also, indicating an epsilon-production via an explicit comment, as shown above, helps readibility.