Friedrich Wilhelm Schröer

The GENTLE Compiler Construction System

Extensions

Contents

1 Introduction **5**

2 Target Tokens **6**

3 Iteration Statements **9**

4 Conditional Compilation **12**

5 Persistent Storage **14**

6 General Grammars **17**

7 Language Interoperability **19**

1 Introduction

The GENTLE Compiler Construction System, originally designed in 1989 at the German National Research Institute for Information Technology, has been used for many years in industry, research, and education.

GENTLE 97 is provided for personal usage, open source projects, and courses on compiler construction. This edition is freely available on the Internet since 1997.

GENTLE 97 is described in the book

```
Friedrich Wilhelm Schröer
The GENTLE Compiler Construction System
R. Oldenbourg Verlag
Munich and Vienna, 1997
```

GENTLE 21 is a commercial edition for industrial projects. It is distributed and supported by Metarga GmbH since 2001.

GENTLE 21 provides a number of extensions:

- Target Tokens
- Iteration Statements
- Conditional Compilation
- Persistent Storage
- General Grammars
- Language Interoperability

These extensions are described in this document.

2 Target Tokens

Whereas source tokens constitue the source text of a compiler, target tokens can be used to define the target text. Just like source tokens appear as strings inside source grammar rules, target tokens may appear as special strings inside rules of unparsing predicates. Unparsing is symmetric to parsing.

This feature makes emit-predicates superfluous (that often obscure the target text) and supports a template-like style of describing the output.

A target token is a sequence of characters enclosed in sharp characters

```
# ... #
```

The enclosed characters are written to the target file. For example

```
#text#
```

emits

```
text
```

to the target file.

Because target tokens often correspond to complete lines of the target text and to avoid newline characters inside these tokens there is the convention that if a target token is not closed on the same line as that of the opening sharp, then it is implicitely closed and a newline is appended to the text. For example

```
#text
```

emits

```
text<newline>
```

The only escape character inside target tokens is the sharp character (#). Thus a backslash just stands for a backslash. For example,

```
#printf("hello world\n");#
```

emits

```
printf("hello world\n");
```

Inside target tokens the following escape sequences may be used;

```
#n#
```

stands for a newline character,

```
##
```

stands for a sharp character.

Variables may appear inside target tokens making a mixture of tokens and emit predicates superfluous. An interspread variable `V` is written in the form

```
#V#
```

For example, if `Size` is a variable with value `256` then

```
#char buffer[#Size#];#
```

emits

```
char buffer[256];
```

(as you may guess from this example, if a target token is follwed by an identifier, it should be separated by white space)

Such variables may be of any type. If the variable of type `INT` the decimal representation is used. If the variable is of type `STRING` the characters that constitue the string are used (without terminating zero and without enclosing quotes).

For all other types of variables that appear inside target tokens the user has to supply an emit routine.

If `T` is the type, the emit routine is called

```
emit_T
```

It takes a value of `T` as argument.

This routine can be implemented as a Gentle action.

For example, if `T` is defined as

```
'type' T
   foo
   bar
```

the emit routine can be specified as

```
'action' Emit_T(T)
'rule' Emit_T(foo)
   #f_o_o#
'rule' Emit_T(bar)
   #b_a_r#
```

The code

```
where(foo -> Foo)
where(bar -> Bar)

#Foo=#Foo#, Bar=#Bar##
```

then emits

```
Foo=f_o_o, Bar=b_a_r
```

Target tokens are written onto the target file. If this file is not explicitely opened by the user, it is stdout.

The target file can be opened by invoking the predicate

```
'action' OpenTargetFile (STRING)
```

and close by invoking

```
'action' CloseTargetFile
```

If the target file is open then OpenTargetFiles causes an implicit preceeding CloseTargetFile

These predicates are implemented in the Gentle runtime system, but nevertheless they need to be declared as an external predicates by the user.

For example,

```
#This text is emitted to stdout
OpenTargetFile("file_1")
#This text is emitted to "file_1"
OpenTargetFile("file_2")
#This text is emitted to "file_2"
CloseTargetFile
```

3 Iteration Statements

Tail-recursive predicates are a common pattern to evaluate a certain sequence of members a number of times.

For example, given the following definitions,

```
'action' Example
'rule' Example
   Traverse( list(1, list(2, list(3, nil))) )

'action' Traverse(LIST)
'rule' Traverse(List)
   [|
      where(List -> list(Head, Tail))
      print(Head)
      Traverse(Tail)
   |]
```

the members

```
where(List -> list(Head, Tail))
print(Head)
```

are evaluated in succession with

```
List = list(1, list(2, list(3, nil))),
List = list(2, list(3, nil)),
List = list(3, nil),
List = nil.
```

This can be expressed more concisely by an iteration statement:

```
'action' Example
'rule' Example
   'iter' List 'first' list(1, list(2, list(3, nil)))
      where(List -> list(Head, Tail))
      print(Head)
   'next' Tail
```

An iteration construct takes the form

```
'iter' VAR 'first' FIRST
   BODY
'next' NEXT
```

where VAR is a fresh variable and FIRST and NEXT are expressions, VAR, FIRST, and NEXT are of the same type. BODY is an arbitrary sequence of members.

FIRST may only use variables that are defined by the left context. NEXT may also use variables that are defined inside BODY.

Variables defined inside BODY are not visible outside the construct. The variable VAR is visible inside and after the construct.

The construct is evaluated as follows:

- The variable VAR is initially defined by the value of FIRST.
- BODY is evaluated. If BODY fails the construct terminates. If BODY succeeds the variables appearing in NEXT are defined, and the variable VAR is redefined by the value of NEXT. This step is repeated until BODY fails.

Example:

In

```
'iter' List 'first' list(1, list(2, list(3, nil)))
   where(List -> list(Head, Tail))
   print(Head)
'next' Tail
```

VAR is

```
List
```

FIRST is

```
list(1, list(2, list(3, nil))),
```

BODY is

```
where(List -> list(Head, Tail))
print(Head)
```

and NEXT is

```
Tail
```

In the first iteration List is defined as list(1, list(2, list(3, nil))). Head is defined as 1 and Tail is list(2, list(3, nil)).

In the second iteration List is defined as Tail, i.e. list(2, list(3, nil)). Head is defined as 2 and Tail is list(3, nil).

In the third iteration List is again defined as Tail, i.e. this time as list(3, nil). Head is defined as 3 and Tail is nil.

In the forth iteration List is defined as nil, hence

```
where(List -> list(Head, Tail))
```

fails and the construct terminates.

An iteration construct may also take the form

```
'iter' VAR1, VAR2 'first' FIRST1, FIRST2
   BODY
'next' NEXT1, NEXT2
```

which is evaluated as follows:

- The variables `VAR1` and `VAR2` are initially defined by the values of `FIRST1` and `FIRST2`, respectively.
- `BODY` is evaluated. If `BODY` fails the construct terminates. If `BODY` succeeds the variables appearing in `NEXT1` and `NEXT2` are defined, and the variables `VAR1` and `VAR2` are redefined by the values of `NEXT1` and `NEXT2`, respectively. This step is repeated until `BODY` fails.

Example:

In

```
'iter' List, Sum 'first' list(10, list(20, list(30, nil))), 0
   where(List -> list(Head, Tail))
'next' Tail, Sum+Head

print(Sum)
```

the variable `List` takes the same values as above, In parallel, the variable `Sum` takes the values `0`, `10`, `30`, `60`. Hence

```
print(Sum)
```

prints `60`.

4 Conditional Compilation

The construct

```
[| Test Code |]
```

can be used to execute Code only if Test succeeds. Here the execution of Code depends on the dynamic behaviour of Test.

Gentle also allows to include or exclude code statically depending on command line flags. Here the code is skipped by Gentle if the flag is not specified.

This can be used to maintain several variants of a product in a single file.

In

```
%{ Flag : Code %}
```

Code is processed if `Flag` is set on the command line using the "`-d`" option: option

```
gentle -d Flag spec.g
```

If `Flag` is not set, `Code` is skipped.

`Flag` is constructed from letters, numbers, and the characters "`_`" and "`-`".

For example,

```
'nonterm' Assignment
'rule' Assignment
   Variable
   %{Pascal: ":=" %} {%C: "=" %}
   Expression
```

can be used to switch between two versions of the assignment operator.

This construct can appear everywhere where a token is allowed, e.g. inside type definitions:

```
'type' Color
   red
   %{ Y : yellow %}
   blue
```

Here `yellow` is only defined as functor of `Color`, if

```
-d Y
```

is specified on the command line.

Variant selectors can be nested:

```
%{ A :
   print("Alice")
   %{ B :
      print("Bob")
   %}
%}
```

If `-d A -d B` is specified, then `Alice` and `Bob` is printed. If only `-d A` is specified, only `Alice` is printed (to be more precisely: code to print `Alice` is generated).

More than one flag can be used:

```
%{Pascal, Modula: ":=" %} %{C, Java: "=" %}
```

If `-d Pascal` or `-d Modula` is specified, the assignment operator is "`:=`", if `-d C` or `-d Java` is specified, the assignment operator is "`=`".

More than one flag can be specified on the command line. For example,

```
gentle -d Pascal -d Debug spec.g
```

switches on all variant selectors that contain the flags `Pascal` and/or `Debug`.

A variant is selected if and only if one of its flags is specified on the command and if it appears globally or inside a selected variant.

A variant selection may be seen as condionally "commenting out" a fragment text. Here is how it interacts with other forms of comments.

In

```
/* %{ Flag : */ %} */
```

the "`%{ Flag :`" is commented out by "`/*`" and "`*/`", regardless wether `Flag` is specified on the command line or not. Hence, "`%}`" has no coressponding "`%{`", and the second "`*/`" has no corresponding "`/*`". I.e., the construct is erroneous.

In

```
%{ Flag : /* %} */
```

the "`/*`" is included if `Flag` is specified on the command line, resulting in a valid (empty) comment "`/* */`". If `Flag` is not specified, the "`/*`" is skipped and the "`*/`" has no corresponding "`/*`", the construct is invalid.

As a rule, "`/* ... */`" and "`%{ ... %}`" should properly nest.

5 Persistent Storage

Gentle data structures may be stored into and retrieved from XML files.

The command line option `-x` as in

```
gentle -x spec.g
```

creates XML readers and writers for the data types defined in `spec.g`

This includes all builtin types `INT`, `STRING` , `POS`, and all kinds of user defined types: terms, table indices and fields, and also opaque types (types that are defined outside the Gentle specification, see below for the handling of these types).

The builtin predicate

```
xmloutput
```

is used to emit an XML representation of its argument

For example,

```
xmloutput(Filename, Value)
```

emits the XML representation of `Value` to file with name `Filename`. The complete value with all constituents is emitted. If a constituents is a table index, then all fields associated with that index are emitted. Cyclic data structures are allowed and are flattened automatically.

The builtin predicate

```
xmlinput
```

is used to read an XML value written with `xmloutput`.

For example,

```
xmlinput(Filename, Typename -> Var)
```

reads a value of type `Typename` from file with name `Filename` and assign it to `Var`. If the value contains table indices the corresponding entries are created from scratch.

Opaque Types

The default reader and writer for opaque types treat them like `INT`. Of course this does not work if the values are references to complex structures. So readers and writers for opaque types can be defined the user.

For example, given the declarations

```
'type' MYTYPE

'action' MyConstructor(-> MYTYPE)

'action' RegisterMyProcs

'action' Demo
'rule' Demo

   RegisterMyProcs

   MyConstructor(-> Q1)
   xmloutput("xmlfile", Q1)
   xmlinput("xmlfile", MYTYPE -> Q2)
   print(Q2)
```

the action `Demo` first registers the user defined reader and writer by invoking the external predicate `RegisterMyProcs`.

The calls to `xmloutput` and `xmlinput` then use the user defined functions.

These can be declared in C like this:

```
#include <stdio.h>
extern FILE *xmlfile;

MyWriter(x)
   long x;
{
   fprintf(xmlfile, "%d\n", x);
}

long MyReader()
{
   long x;

   fscanf(xmlfile, "%d", &x);
   return x;
}
```

The registration procedure looks like this:

```
RegisterMyProcs()
{
   extern (*userxmlwriter_MYTYPE)();
   extern long (*userxmlreader_MYTYPE)();

   userxmlwriter_MYTYPE = MyWriter;
   userxmlreader_MYTYPE = MyReader;
}
```

6 General Grammars

Instead of Yacc ("Yet another compiler compiler") now also Accent ("A compiler compiler for the entire class of context free grammars") can be used. Whereas Yacc requires the grammar to be LALR(1), Accent does not impose any restrictions.

When implementing a language that is defined by a standard document, Yacc is often not capable to process the original grammar. Hence this has to be adapted to the LALR(1) restrictions - an error-prone process. Accent can process the original grammar.

When designing a new language, one can avoid struggling against LALR(1) and start with the natural grammar that can be used as the reference definition. If the language becomes stable, the implementation can migrate to Yacc where only the critical parts of grammar have to be changed. However, Accent has turned out to be a true alternative to Yacc and it is used in production compilers.

To use Accent specify `-a` in the command invoking Gentle:

```
gentle -a spec.g
```

Accent can even process ambiguous grammars. See the Accent documentation for a discussion how to resolve ambiguities by grammar annotations.

The Accent annotations

```
%prio N , %short , %long
```

are written in Gentle as

```
'prio' N , 'short' , 'long'
```

respectively.

Note that the tool Reflex is not aware of Accent. Because the default `YYSTYPE` block of Reflex is Yacc specific, Gentle generates an Accent specific file `YYSTYPE.b` when invoked with option `-a`.

Here is a script that generates a compiler using Gentle, Lex, and Accent:

```
ACCENTHOME=$HOME/accent
ACCENT=$ACCENTHOME/accent/accent
ART=$ACCENTHOME/art/art.o

GENTLEHOME=$HOME/gentle
GENTLE=$GENTLEHOME/gentle/gentle
GRTS=$GENTLEHOME/gentle/grts.o
REFLEX=$GENTLEHOME/reflex/reflex
LIB=$GENTLEHOME/lib
```

```
LEX=lex
CC=cc

$GENTLE -a spec.g

$REFLEX YYSTYPE=AccentYYSTYPE.b

$LEX gen.l

$ACCENT gen.y

$CC -o program \
   spec.c \
   lex.yy.c \
   yygrammar.c \
   $LIB/errmsg.c \
   $LIB/main.o \
   $ART \
   $GRTS
```

Accent can be obtained from Metarga.

7 Language Interoperability

If Gentle is used to build a compiler, i.e. a program that reads a source text and writes a target file, there is rarely a need to escape to other languages. However, if Gentle is used to build an application where a source text results in an internal data structure that is then processed by modules that must be written in a foreign language, the situation is different. (For example, in an industrial project Gentle was used to compile the Security Policy Specification Language SPSL into an internal policy representation. This representaion was then processed by a firewall that had to be written in C++ because existing C++ libraries, e.g. for Internet Key Exchange, had to be used.)

Gentle can generate header files that allow to access and manipulate Gentle terms as C++ objects.

This is specified by the option `-h` on the command line.

Assume that `spec.g` contains the following type definition:

```
'type' Expr
   plus (x: Expr, y: Expr)
   mult (x: Expr, y: Expr)
   num (n: INT)
```

Then

```
gentle -h spec.g
```

creates the additional files

```
spec.f
spec.h
spec.sig
```

which are described in the following.

spec.f

A header file that defines type `Expr` without giving details of the representation. This file is required to support mutually recursive type definitions and serves as a forward declaration

```
typedef struct Expr_struct *Expr;
```

spec.h

A header file that defines type `Expr` and its subtypes

```
struct Expr_struct {
   int tag;
};
typedef struct Expr_struct *Expr;
```

For each functor, e.g. `plus`, a subtype is introduced

```
struct plus_struct : Expr_struct {
   Expr x;
   Expr y;
};
typedef struct plus_struct *plus_subtype;
```

The tagfield of `Expr` is set `plus_tag` which is defined as a constant

```
#define plus_tag 1
```

There is also a constructor functions that allows to create `plus`-terms

```
extern "C" plus_struct *plus (
   Expr x,
   Expr y
);
```

spec.sig

This is the signature of type `Expr` in the format of the tool Memphis (see below).

Usage

In a C++ program include the generated header files to make the data types available

```
#include "spec.f"
#include "spec.h"
```

You may declare variables of Gentle types

```
Expr e;
```

You may construct terms using the generated constructor functions:

```
e = plus(mult(num(1),num(2)),num(3));
```

You can access the tag field to inspect the which kind of value has been assigned to an Expr variable

```
if (e->tag == plus_tag) {
   /* e is a value of the form plus(...) */
}
```

You can declare variables of a subtype (e.g. for values with functor `plus`) and assign Expr values to them (if the `Expr` variable dynamically holds a plus value)

```
   plus_subtype e_plus = (plus_subtype) e;
```

This allows access to the fields of the plus term using the "`->`" notation

```
e_plus->x
e_plus->y
```

For example, the Gentle predicate

```
'action' ReversePolish(Expr)
   'rule' ReversePolish(plus (X, Y)):
      ReversePolish(X)
      ReversePolish(Y)
      print("plus")
   ...
```

could be written in C++ in this way

```
void ReversePolish(Expr e)
{
   if (e->tag == plus_tag) {
      plus_subtype e_plus = (plus_subtype) e;

      ReversePolish(e_plus->x);
      ReversePolish(e_plus->y);
      printf("plus\n");
   }
   ...
}
```

Memphis

The tool Memphis adds Gentle style pattern matching to C++.

Using this tool the above function could have been written in this way

```
void ReversePolish(Expr e)
{
   match e {
      rule plus (x, y) :
         ReversePolish(x);
         ReversePolish(y);
         printf("plus\n");
      ...
   }
}
```

Memphis can be obtained from Metarga.

Field Names

If your Gentle specification does not use names for the fields of a functor, e.g. if, instead of

```
'type' Expr
   plus (x: Expr, y: Expr)
   ...
```

you have witten

```
'type' Expr
   plus (Expr, Expr)
   ...
```

then field names are generated as if you would have written

```
'type' Expr
   plus (field1: Expr, field2: Expr)
   ...
```

Qualified Subtypes Names

Gentle allows to use the same functor name in different types. If your type definitions rely on this feature you have to to use the `-q` option when invoking Gentle (the option `-q` implies the option `-h`).

This generates subtype names (functor names) that are qualified with the name of the base type

For example, The subtype

```
plus_subtype
```

is named

```
Expr_plus_subtype
```

The constructor function

```
plus(...)
```

is named

```
Expr_plus(...)
```

The tag value

```
plus_tag
```

is named

```
Expr_tag
```