Grammar Deployment Kit Reference Manual

Jan Kort[1]
[1] Universiteit Amsterdam

May 2, 2003

Contents

Abstract

Grammar deployment is the process of turning a given grammar specification into a working parser. The *G*rammar *D*eployment *K*it (for short, GDK) provides tool support in this process based on grammar engineering methods. We are mainly interested in the deployment of grammars for software renovation tools, that is, tools for software re- and reverse engineering. The current version of GDK is optimized for Cobol. We assume that grammar deployment starts from an initial grammar specification which is maybe still ambiguous or even incomplete. In practice, grammar deployment binds unaffordable human resources because of the unavailability of suitable grammar specifications, the diversity of parsing technology as well as the limitations of the technology, integration problems regarding the development of software renovation functionality, and the lack of tools and adherence to firm methods for grammar engineering. GDK helps to largely automate grammar deployment because tool support for grammar adaptation and parser generation is provided. We support different parsing technologies, among them `btyacc`, that is, `yacc` *with backtracking*. GDK is free software.

Chapter 1

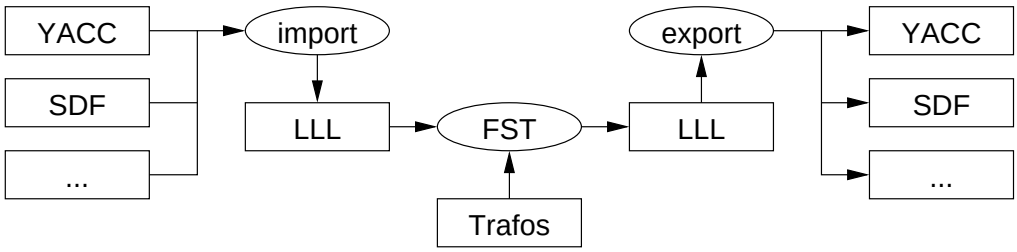
Structure of GDK

The Grammar Deployment Kit (GDK) is a lightweight ANSI-C-based kit for getting from a grammar specification to a parser suitable for automated software renovation. Developing renovation parsers for Cobol or any other complex real-world language, is not trivial because of complicated syntax rules, the variety of existing dialects and embedded languages, or pre- and post-processing issues [1]. GDK is based on a few principles and assumptions: (i) recovery of base-line grammars from resources like language references or compilers [3], (ii) grammar adaptation by automated transformations to enable grammar manipulation in a traceable manner [2], (iii) parser generation for various parsing technologies, (iv) selection of a suitable grammar format for grammar engineers, (v) focus on the grammar part of a renovation parser as opposed to idiosyncratic scanners, pre- and post-processors. The current version of GDK is optimized for the deployment of Cobol grammars.

GDK *components*

- LLL: a simple EBNF-based grammar format.
- LLLIMPORT: a tool to import grammars.
- LLLEXPORT: a tool to export grammars.
- FST: a tool to transform grammars.
- TT: a library for parsing, transformation, and unparsing.
- VS COBOL II: deployment of a Cobol grammar for software renovation.

We will discuss all these components accordingly. In the VS COBOL II section, we will also sketch an illustrative renovation tool EXPAND dealing with data expansion. This sort of software renovation task is similar to problems a la Y2K or Euro conversion.



Chapter 2

Getting and Installing GDK

The GDK distribution can be obtained from `http://gdk.sourceforge.net`. The distribution can be unpack and installed using the following commands:

```
gzip -d gdk-1.1.tar.gz
tar -xf gdk-1.1.tar
cd gdk-1.1
./configure
make
make check
make install
```

The configure command can be given options to guide the installation, the most useful options are:

- `--prefix=<dir>` The directory where to install GDK .
- `--enable-bison` Enable bison support.
- `--enable-sglr` Enable sglr support.

Chapter 3

The LLL grammar format

LLL should be read as "L-Cube". The name hints on *L*ightweight LL parsing which is the built-in parsing technology of GDK. The LLL grammar format crosscuts all components in GDK in that it is used as (i) the primary format to import recovered grammar specifications for deployment, (ii) to develop new grammars, (iii) to transform grammars in the course of deployment, and (iv) to generate parsers for various technologies.

EBNF of LLL *grammar format in* LLL *notation*

```
specification : rule+;
rule          : ident ":" disjunction ";";
disjunction   : {conjunction "|"};
conjunction   : term+;
term          : basis repetition?;
basis         : ident
              | literal
              | "%epsilon"
              | alternation
              | group
              ;
repetition    : "+" | "*" | "?";
alternation   : "{" basis basis "}" repetition;
group         : "(" disjunction ")" ;
```

The top `rule`'s identifier is called `specification` and consists of a list of `rule`s. A `rule` has an identifier which is unique to this rule and a `disjunction`. Not all identifiers need to have a rule however, if an identifier does not have a rule it is called a *bottom* identifier, which can be either a lexical token or a part of the grammar defined outside the current grammar (see the section on namespaces). The `disjunction` is a list of `conjunction`s separated with "|". A `conjunction` is a list of `term`s. A `term` consist of a `basis` and an optional `repetition` symbol: "+" for lists containing one or more elements, "*" for lists containing zero or more elements and "?" for optionals. The `basis` can be an identifier, a literal (i.e. keyword or symbol), `%epsilon`, `alternation` (also called separator lists) or `group`. The `alternation` denotes an alternation between two `basis`. A typical example of an alternation is a paramter lists, e.g.:

```
params: {param ","}+;
```

Finaly, the `group` allows for nesting `disjunction`s. The presence or absence of groups is an important concept in GDK , if there are no groups the grammar is considered simple. A simple grammar makes it easy to generate sensible parsers with meaningful names for parsing systems that do not support nesting. Mapping a nested grammar to a system that does not support nesting involves generating dummy names for the nested constructs, this means either generating very long names (e.g. flattening a conjunction of idents by concatenating all the names) or inventing names that have no relation to the original construct (e.g. phrase1, phrase2, etc.). Either way the generated parser will be easier to debug if every construct has a sensible name.

TODO: need some general format to present examples. Should all examples speak for themselves ? Should they be figures with some caption ? Some box around them maybe ?

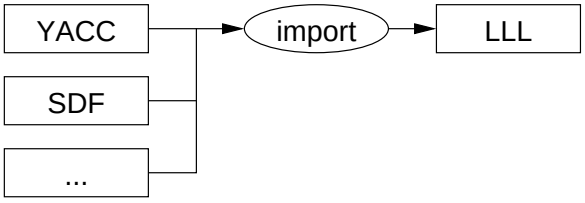
A VS Cobol II *sample: the* `MOVE`*-statement*

```
move-statement
  : move-statement-1
  | move-statement-2
  ;

move-statement-1: "MOVE" cobword-lit "TO" identifier+;
move-statement-2: "MOVE" corresponding identifier "TO" identifier+;
```

A suitable grammar format is crucial for grammar deployment. There are three major problems with commonly used grammar formats. (i) A too restrictive format such as the simple BNF format underlying `yacc` requires encoding of lists and optionals. (ii) On the other hand, use of a too liberal format, e.g., non-trivially nested phrases makes the grammar less suitable for matters of abstract syntax, debugging, and adaptation. (iii) Using the input language of a specific parser generator for grammar recovery, development, maintenance, and others enforces one to deal with the idiosyncrasies of the format all the time. LLL supports EBNF to enable *regular expression operators* in the view of (i), but it *restricts* them to rule out (ii). This is achieved by simply disallowing the operator `group`. The reason for having this operator at all is to be able to import grammars from different formats. In the first FST transformation script all groups will have to be removed because at the end of each script left over group operators are reported as errors. Furthermore, LLL is a *pure* EBNF notation to remedy (iii), that is, LLL does not directly deal with conflict resolution, disambiguation, lexical syntax, and others. Our experience with many grammar recovery, development, and deployment projects resulted in a simple and sound grammar notation that is particularly suited for developing renovation parsers.

Chapter 4

LLLimport



WORK IN PROGRESS.

The LLLIMPORT tool is used to transform an external grammar to an LLL grammar. The format of the command is:

```
lllimport -f <external-format> < <external-grammar> > <lll-grammar>
```

For example:

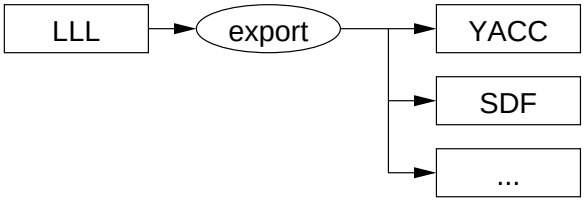
```
lllimport -f sdf < COBOL.sdf > COBOL.lll
```

4.1 YACC

4.2 SDF

Chapter 5

LLLexport



The LLLexport tool is used to transform an LLL grammar to an external grammar. The format of the command is:

```
lllexport [options] < <lll-grammar> > <external-grammar>

Usage: lllexport [options] <spec.lll> ...
Options:
  -h            Show usage
  -c <ident>    Place a BTYACC cut (YYVALID) on <ident>
  -s <ident>    Skeleton grammar start symbol
  -r <ident>    Identifier for rejects
  -f <format>   Output Format: use -f help to get a list
  -p <prefix>   Prefix for generated names, default: FOO
  -q <module>   Default module name, default: Junk
  -e <module>   External module
  -y <option>   Yacc option, e.g. -y "%glr-parser"
  -m            Generate multiple SDF files
```

For example:

```
lllexport -f sdf COBOL.lll > COBOL.sdf
```

For some formats, such as `sdf` (without `-m`), one file will be generated. For other formats, such as `c` multiple files will be generated, so the output redirection is not used and the prefix specified with `-p` will be used to make sure there are no filename conflicts.
To simplify Makefile writing, each call to LLLexport will generate exactly one file. The exception is SDF, the `-m` option can be used to generate multiple files which can be used with the ASD+SDF meta environment, this is an interactive enviroment so correct Makefile dependencies are not an issue.
Because the LLL format says nothing about lexical constructs the user has to supply a lexer. Because writing a lexer for a large language such as COBOL is a lot of work, LLLexport provides some support to make the job easier. The main gain is achieved by generation of code having to do with keywords, either recognition of keywords in the case of a scannerful approach or keyword follow restrictions and identifier rejections in the case of the scannerless aproach. Perhaps more subtle is the gain achieved by the generation of a template, although the user is expected to be familiar with the parsing system code is being exported to, it is still useful to have a full template rather than a list of lexicals that have to be implemented.

5.1 YACC

Support for YACC based tools, at the moment only BTYACC works reliably. Standard Bison is not sufficient since it only supports LALR(1), although the GLR mode works to some extend it doesn't parse as much as BTYACC for unknown reasons. The `-f yacc` option of LLLexport can be used to generate all the relevant files.

5.2 SDF

SDF is a scannerless grammar format and requires the user to specify which identifiers have to be rejected when a keyword is recognized, this can be done with the option `-r <ident>`.

TODO: more intro, show SDF code, add references etc.

5.3 C Based Combinator Parsing

For example, for the move statement the following code would be generated:

```
TTterm
TTparseVSC_stat_move_statement_1( void )
{
    fparse fs[4];

    fs[0] = TTparset_MOVE;
    fs[1] = TTparseVSC_stat_id_or_lit;
    fs[2] = TTparset_TO;
    fs[3] = TTparseVSC_data_ref_identifier_plus;
    return TTparseVSC_CONJUNCTION(TTfunVSC_stat_move_statement_1, fs);
}
```

5.4 Haskell Based Combinator Parsing

With `-f haskell` a Haskell combinator parser based on ParseLib will be generated as well as a datatype representing the AST. After this Gmap can be used to transform or analyze the parsetree, it is expected that Gmap will soon be part of the GHC distribution.

5.5 Term Matching and Construction

Three different kind of functions will be generated:

- `TTis` followed by the name of the term that is going to be typechecked. The typechecking is shallow, for example for a list of statements, it will only be checked wether it is a list and weather the head of the list is a statement.
- `TTbuild` followed by the name of the term that is created and n term arguments, where n is the arity of the term. A typecheck is performed on the arguments before the term is created.
- `TTmatch` followed by the name of the term that is going to be match upon, the first argument of the function will be the term being matched upon followed by n arguments where n is the arity of the term.

For example, for the move statement the following code would be generated:

```
BOOL
TTisVSC_stat_move_statement_1(TTterm t)
{
        return t->fun == TTfunVSC_stat_move_statement_1;
}
```

```
TTterm
TTbuildVSC_stat_move_statement_1(TTterm t1, TTterm t2, TTterm t3, TTterm t4)
{
    assert(TTist_MOVE(t1));
    assert(TTisVSC_stat_id_or_lit(t2));
    assert(TTist_TO(t3));
    assert(TTisVSC_data_ref_identifier_plus(t4));
    args[0] = t1;
    args[1] = t2;
    args[2] = t3;
    args[3] = t4;
    return TTbuild_term(TTfunVSC_stat_move_statement_1, args);
}

BOOL
TTmatchVSC_stat_move_statement_1(TTterm t,
        TTterm *t1, TTterm *t2, TTterm *t3, TTterm *t4)
{
    if (t->fun == TTfunVSC_stat_move_statement_1)
    {
        *t1 = t->u.args[0];
        *t2 = t->u.args[1];
        *t3 = t->u.args[2];
        *t4 = t->u.args[3];
        return TRUE;
    }
    return FALSE;
}
```

Chapter 6

GDKconfig

The GDKconfig tool is used to aquire information about the installment of GDKThe format of the command is:

```
gdk-config [options]
```

Legal options are:

- `--version` The version of GDK that is installed.
- `--cflags` The C compiler flags needed when building C based applications.
- `--libs` The linker flags needed when linking C based applications. The `tt` library will always be linked, other possible libraries are: `lll`, `sdf`, `fst` etc.

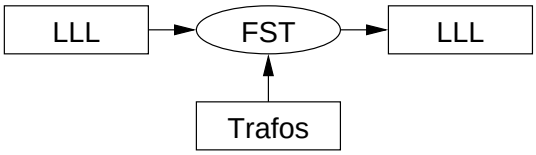
The idea is to use these flags in a Makefile to make it independend on where GDK is installed. For example:

```
INC=‘gdk-config --cflags‘
LIB=‘gdk-config --libs‘

foo.o:foo.c
        gcc -c foo.c $(INC)

foo:foo.o
        gcc -o foo foo.o $(LIB) -lsdf
```

This will compile and link `foo.c` against the bison based VS COBOL II parser.

Chapter 7

FST



The FST tool is used to transform grammars. The format of the command is:

```
fst <script.trafo> <spec.lll> > <grammar-out>
```

For example:

```
fst naming.trafo Spec.lll > Named.lll
```

Would take the grammar `Spec.lll` and a transformation script `naming.trafo` and generate the grammar `Named.lll`.

7.1 FST Operators

There are 16 operations available to transform grammars:

```
%delete     ident
%eliminate  ident
%reject     ident
%rename     ident %to ident
%unify      ident %to ident
%separate   ident %to ident
%introduce  rule
%include    rule
%exclude    rule
%resolve    rule
%preserve   rule
%restrict   rule
%generalize rule
%fold       rule
%unfold     ident
%redefine   rule %to rule
%write      literal
```

`%delete ident`

Delete the identifier `ident` from all rules, i.e. substitute `ident` with `%epsilon`. A simplifier will get rid of any redundant epsilons. For example doing `%delete c` in:

```
a : b c* d;
```

will give the intermediate result:

```
a : b %epsilon* d;
```

which will be simplified to:

```
a : b d;
```

Not all epsilons can be removed however. For example doing `%delete c` in:

```
a : b | c | d;
```

will result in:

```
a : b | %epsilon | d;
```

This will not be simplified further. If the wanted result is:

```
a : b | d;
```

the user will have to explicitly obtain this by adding a second transformation:

```
%exclude a : %epsilon;
```

`%eliminate ident`

Eliminate the rule identified by `ident`. After elimination `ident` should be fresh.

`%reject ident`

Reject the rule identified by `ident`. If `ident` still exists somewhere after the reject operation it will be a bottom identifier now.

`%rename ident1 %to ident2`

Rename the identifier `ident1` to `ident2`. The identifier `ident1` should exists and the identifier `ident2` should not exist prior to the renaming.

`%unify ident1 %to ident2`

Unify the bottom identifier `ident1` to the existing identifier `ident2`.

`%separate ident1 %to ident2`

Separate the rule `ident1` from the grammar by replacing `ident1` to `ident2` in the right hand sides of every rule.

`%introduce rule`

Introduce a new rule.

`%include rule`

Add the disjunction of `rule` to the existing one.

`%exclude rule`

Subtract the disjunction of `rule` from the existing one.

`%resolve rule`

Provide a definition for a bottom `identifier`.

`%preserve rule`

Redefine an existing rule with a syntactically equivalent definition.

`%restrict rule`

Redefine an existing rule with a definition that is a syntactical restriction.

`%generalize rule`

Redefine an existing rule with a definition that is a syntactical generalization.

`%redefine rule %to rule`

Give a new definition for a rule. This is a dangerous operation and should only be used in cases where gradual grammar changes would get too complex. To at least provide some safety, the original rule has to be provided.

`%fold rule`

Fold nested constructs to identifiers. For example,

```
qualified-name : data (("IN"|"OF") library)*;

%fold in-library : ("IN" | "OF") library;

qualified-name : data in-library*;
in-library : ("IN" | "OF") library;
```

This is useful for making the grammar more readable, especially if there are many deeply nested constructs.

`%unfold rule %to rule`

Unfold identifiers to nested constructs. One example would be to undo the fold above, but a more realistic example is to convert BNF constructs to EBNF constructs:

```
func: id "(" params ")";
params
  : param
  | param "," params
  ;

%preserve params: {param ","}+;
%unfold params

func: id "(" {param ","}+ ")";
```

7.2 FST and Namespaces

Namespacing is achieved by explicitly qualifying every identifier. This means that to put something in a certain namespace is nothing more than a `\%rename`. What a namespace means exactly is dependend on the system code is generated for. Some targets like ANSI C do not have namespacing, in wich case names will become long. Others have a module system, like Haskell. There are some conventions that can be used however, for example:

```
%rename move-statement %to VSC/stat/move-statement
```

This will place the `add-statement` in the package `VSC` and the module `stat`. Using modules for namespaces as in the example below is not a good idea.

```
%rename add-statement %to VSC/stat/add
```

Some module systems will not report name conflicts. Rather they will merge different definitions of the same name in different modules.

Chapter 8

Transformation Technology

GDK is a self-contained kit for grammar deployment. This means one is not required to employ any third-party components to develop simple software renovation tools. In fact, the C-based TT offers lightweight functionality for parsing, parse-tree construction, rewriting, traversal, unparsing, and pretty printing. Hence, there is no need to install additional software to assess the suitability of the other components of GDK, especially the VS COBOL II grammar which was deployed with GDK. As an aside, TT is also used for the implementation of FST, LLLEXPORT and LLLIMPORT.

Modules in GDK *V1.0 (2059 LOC in total)*

Module	Intention	LOC
`ttterm.c`	Term construction and inspection	229
`ttutils.c`	Traversal, container, unparsing functionality	1019
`ttstrat.c`	Strategies	300
`ttparse.c`	Combinator parsing	286
`ttscan.c`	Simple scanner for combinator parsing	98
`pretty.c`	Pretty printing (TODO: port this)	
`tttoken.c`	Tokenization	78
`ttstack.c`	Generic stacks	49

The TT functionality can be used for the implementation of simple renovation tools. That is, TT is complementary to more sophisticated technologies such as ASF+SDF, DMS, or REFINE. To give examples of current limitations of TT, the term format `MTerm` provided by the TT does not support garbage collection, and the C-encodings of rewrite rules and traversals are not type-checked whereas this is suported, e.g., by ASF+SDF. On the other hand, for TT it is not necessary to learn a new language, which is the case for more sophisticated tools. The performance of TT-based combinator parsers is only outperformed by `btyacc`. Here, we take advantage of grammar class restrictions for combinator parsing. These restrictions are not necessarily appropriate for all languages.

In the rest of this chapter the most interesting parts of the transformation library will be discussed in more detail.

8.1 Terms

A term consists of a function symbol and arguments. The function symbol contains all kinds information about the arguments: the type of the arguments (term, string or integer), in case the type of arguments is term the function symbol also says how many arguments the term has (also called the arity). There is a special kind of term called a strategy, in this case the function symbol also contains a pointer to the strategy application function.

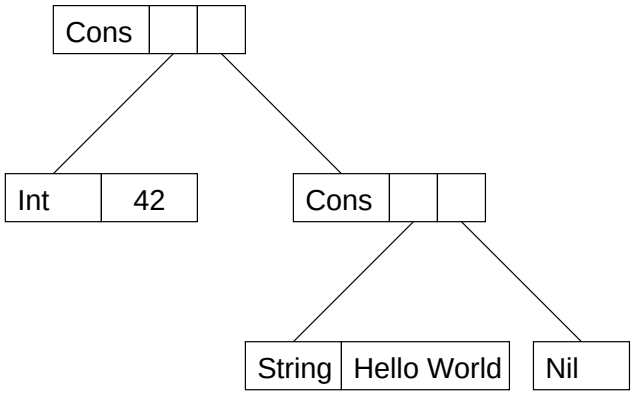
The term structure

```
struct TTterm_struct {
    TTfun fun;
    union {
        int val;
        TTterm args[1];
```

```
        char str[1];
    } u;
};
```

The function symbol `fun` determines which part of the `union` is chosen. Terms can be leafs: `int` and `str` or nodes: `args`. The array declarations `args[1]` and `str[1]` are fake, they are in fact arrays of arbitrary length, this is a well known trick in C programming.

Example term, a list containing two elements: the integer "42" and the string "Hello World"



Construction of example term

```
xs = TTbuild_nil();
xs = TTbuild_cons(TTbuild_string("Hello World"), xs);
xs = TTbuild_cons(TTbuild_int(42), xs);
```

The rest of this section will present the signature of each function in the term library followed by an explanation and an example of its use.

`TTterm TTbuild_term(TTfun fun, const TTterm * args)`

This function creates a term with function symbol `fun` and arguments `args`. The function symbol contains the number of arguments. The argument array is usually a static variable that is reused for more several builds.

```
static TTterm args[1000];

TTterm TTbuild_cons(TTterm hd, TTterm tl)
{
    args[0] = hd;
    args[1] = tl;
    return TTbuild_term(TTfun_cons, args);
}
```

`TTterm TTalloc(TTfun fun)`

This function creates a term with function sumbol `fun` but the arguments are not filled and the term is not yet given to the term library. In many cases `TTbuild_term` is sufficient, but sometimes allocation of memory for a term and giving the term to the term library has to be separated. For example, when doing a type preserving transformation the arguments of the to be created term are gradually filled in a recursive function, this makes using a static array for the arguments to `TTbuild_term` impossible. To avoid the user having to resort to using an explicit malloc, it is more convenient to provide support for separate allocation and the actual creation of the term inside the term library.

```
TTterm TTtransform(TTterm t, TTrewrite_f f)
{
    TTterm result;
    TTfun fun;
    size_t i;

    result = f(t);
    if (result) {
        return result;
    } else if (TTis_node(t)) {
        fun = t->fun;
        result = TTalloc(fun);
        for (i = 0; i < fun->arity; i++) {
            result->u.args[i] = TTtransform(t->u.args[i], f);
        }
        if (equal(result, t)) {
            TTfree(result);
            return t;
        }
        return TTfinish(result);
    }
    return t;
}
```

`void TTfree(TTterm t)`

This function will free the memory allocated by a call to `TTalloc`. This is useful when halfway during the creation of a term it turns out that the result is not needed, or in the case of a type preserving transformation if the term is exactly the same as the original term.

See `TTalloc` for an example.

`TTterm TTfinish(TTterm t)`

This function will give the term to the term library and return the finished term to the user. In the current implementation this function just does a `return t`, but if the term library implemented sharing for example, that would be done inside this function. See `TTalloc` for an example.

`TTfun TTbuild_fun(const char *name, size_t arity)`

This function builds a function symbol with a certain name (useful for debugging) and a certain arrity.

```
void TTutils_init(void)
{
    ...
    TTfun_nil = TTbuild_fun("nil", 0);
    TTfun_cons = TTbuild_fun("cons", 2);
    ...
}
```

`TTfun TTbuild_sfun(const char *name, size_t arity, TTapply_f apply)`

This function builds a function symbol with a certain name and arity like `TTbuild_fun`, but also a strategy application function. See the see section on strategies for more details.

```
static TTterm apply_succeed(const TTstrat * s,
                            TTterm t,
                            const TTstrat_env env)
{
    if (env) return env->empty;
    return t;
}
```

```
void TTstrat_init(void)
{
    ...
    TTfun_succeed = TTbuild_sfun("succeed", 0, apply_succeed);
    ...
}
```

`BOOL TTequal(TTterm t1, TTterm t2)`

This function tests two terms for deep equality.

```
    ...
    if (TTequal(t1, t2)) {
        ...
    }
    ...
```

`TTterm TTbuild_int(int n)`

This function builds a term with the integer function symbol and the integer value `n` as its argument.

```
    ...
    t = TTbuild_int(123);
    ...
```

`BOOL TTmatch_int(TTterm t, int *x)`

This function tries to match an integer: if the term `t` has a function symbol of type integer the variable `x` is set to its value and `TRUE` is returned, otherwise `FALSE` is returned and `x` is unchanged.

```
    ...
    if (TTmatch_int(t, &n)) {
        printf("The number is %d\n", n);
    }
    ...
```

`TTtype_string TTbuild_string(const char * str)`

This function builds a term with the string function symbol and the string `str` as its argument.

```
    ...
    t = TTbuild_string("Hello");
    ...
```

`BOOL TTmatch_string(TTtype_string t, const char ** x)`

This function tries to match a string: if the term `t` has a function symbol of type string the variable `x` is set to point to the argument string and `TRUE` is returned, otherwise `FALSE` is returned and `x` is unchanged.

```
    ...
    const char * str;
    ...
    if (TTmatch_string(t, &str)) {
        printf("The string is: %s\n", str);
    }
    ...
```

`TTtermTTalloc_string(int len)`

This function allocates a term with room to contain a string of length `len`, but does not fill the string yet. This looks similar to `TTalloc` and the reason for having this function is similar too. When creating strings often the length of the resulting string is not known at compile time, this means that the user would have to use malloc and then pass the malloced string to `TTbuild_string` which would then do another malloc. It is better to support the malloc in the term library to avoid this double allocation.

```
TTterm TTto_upper(TTterm t)
{
    int i;
    int len = TTstrlen(t);
    TTterm result = TTalloc_string(len);

    for (i = 0; i < len; i++) {
        result->u.str[i] = toupper(t->u.str[i]);
    }
    return TTfinish_string(result);
}
```

`void TTfree_string(TTterm t)`

This function frees the term `t` which was allocated using `TTalloc_string`. Although there does not seem to be much use for this function it has been included for completeness.

`TTterm TTfinish_string(TTterm t)`

This function will give the term to the term library and return the finished term to the user. In the current implementation this function just does a `return t`. See `TTalloc_string` for an example.

8.2 Scannerless Combinator Parsing

The main asset of GDK is that it is self contained, this is why there is a simple parsing technology included in the distribution called Scannerless Combinator Parsing.

True combinator parsing as in for example Haskell is not so easy to achieve in a low level language like C, the main problem is that C does not allow functions as a return argument. But it is quite straightforward to have one level combinators that take parsing functions and produce a parsetree. This way a small but complex part of the code dealing with the actual parsing can be contained in a small library while the bulk of the parsing code can be generated (from the LLL format). This generated code will be easy to debug and maintain simply because all the complex logic has been taken out and put in the small library. For example the move statement would look like this:

```
TTterm
TTparseVSC_stat_move_1( void )
{
    fparse fs[4];

    fs[0] = TTparset_MOVE;
    fs[1] = TTparseVSC_stat_id_or_lit;
    fs[2] = TTparset_TO;
    fs[3] = TTparseVSC_data_ref_identifier_plus;
    return TTparseVSC_CONJUNCTION(TTfunVSC_stat_move_1, fs);
}
```

There is no complex logic in this code, no if statements, no loops, just plain filling of arrays and calling the appropriate combinator: disjunction or conjunction. The nice thing about this is that the generated code is still readable as opposed to for example C code generated by YACC.

The `TTparseVSC_CONJUNCTION` takes a function symbol and a list of parsers to parse the terms in the conjunction and it results in a term with that function symbol on a succesful parse or 0 to denote failure.

As can be seen in the full list of all the combinators below, each construct in the LLL grammar has a corresponding combinator.

```
TTterm TTparse_disjunction(TTfun fun, size_t n, TTterm(**parsers) (void));
TTterm TTparse_conjunction(TTfun fun, TTterm (**parsers)(void));
TTterm TTparse_plus(TTterm (*parser)(void));
TTterm TTparse_star(TTterm (*parser)(void));
TTterm TTparse_optional(TTterm (*parser)(void));
TTterm TTparse_alternation(TTterm (*p1)(void), TTterm (*p2)(void));
```

`TTterm TTparse_disjunction(TTfun fun, size_t n, TTterm(**parsers) (void))`

This function tries to parse a disjunction. Each parser in the `parsers` array is applied till one is succesful. If none of the parsers is succesful the result will be failure. If there is a succesful parse, a term with function symbol `fun` is constructed, the arity of this function symbol is 1.

```
TTterm
TTparse_statement( void )
{
    fparse fs[3];
    fs[0] = TTparse_move_statement;
    fs[1] = TTparse_if_statement;
    fs[2] = TTparse_perform_statement;
    return TTparse_disjunction(TTfun_statement, 3, fs);
}
```

`TTterm TTparse_conjunction(TTfun fun, TTterm (**parsers)(void))`

This function tries to parse a conjunction. The parsers in the `parsers` array are applied in sequence, each parser consuming part of the input stream. If one of the parsers fails, the result is failure. If all parsers succeed a term with the function symbol `fun` will be constructed, the arity of the function symbol is the same as the number of parsers in the conjunction. The idea is that no information is lost, keywords (and layout) will appear in the parsetree.

```
TTterm
TTparse_move_statement( void )
{
    fparse fs[4];

    fs[0] = TTparse_MOVE;
    fs[1] = TTparse_id;
    fs[2] = TTparse_TO;
    fs[3] = TTparse_id_plus;
    return TTparse_conjunction(TTfun_move_statement, fs);
}
```

`TTterm TTparse_plus(TTterm (*parser)(void))`

This function tries to parse a list of one or more elements. If no element can be parsed the result is failure, otherwise the result is a list of as many parsed terms as possible.

```
static TTterm
TTparse_id_plus( void )
{
    return TTparse_plus(TTparse_id);
}
```

`TTterm TTparse_star(TTterm (*parser)(void))`

This function parses a list of zero or more elements. It will always succeed: if there are no elements to be parsed it will return the empty list.

```
static TTterm
TTparse_id_star( void )
{
    return TTparse_star(TTparse_id);
}
```

`TTterm TTparse_optional(TTterm (*parser)(void))`

This function parses an optional. It will always succeed: if the supplied parser fails it will return the empty list. If it succeeds it will return a singleton containing the parsed term.

```
static TTterm
TTparse_id_opt( void )
{
    return TTparse_opt(TTparse_id);
}
```

`TTterm TTparse_alternation(TTterm (*p1)(void), TTterm (*p2)(void))`

This function parses an alternation. First `p1` is tried, if it fails the result is failure. Then sequences of `p2` followed by `p1` are tried as much as possible. This construction is useful for encoding things like comma separated lists.

```
static TTterm
TTparse_params( void )
{
    return TTparse_alternation(TTparse_id, TTparse_COMMA);
}
```

The thing that is still missing from this picture is the actual scanning, i.e. how to recognize bottom identifiers and keywords. On the one hand it is trivial because they are parsing functions like everything else:

```
TTterm TTparseVSC_KEYWORD(const char * name);
TTterm TTparseVSC_lex_cobword( void );
```

On the other hand there are some subtle issues with the scannerless approach. For example, when doing:

```
TTparseVSC_KEYWORD("MOVE")
```

it is not just a matter of scanning through the buffer for the characters “M”, “O”, “V”, “E” and then stop. There also needs to be a check wether at the current position in the buffer an identifier can be parsed in which case the parsing of the keyword will return false. When recognizing cobol words the opposite applies, for example if the identifier at the current point is identical to “MOVE” it should not be recognized as an identifier. Another issue is that although one can write down any kind of complex lexical recognizer in C it is not very convenient for writing down simple regular expressions. However, having a complete parser in C without dependencies on tools such as flex has definite advantages.

8.3 Strategies

One of the side goals in GDK is to be a showcase for all kinds of transformation technology, to allow users to try out different approaches to program transformation without having to download and install a large ammount of packages.

Strategies were first introduced in (TODO: need reference). TODO: Need some strategies intro.

One thing to bear in mind for the next subsections is that strategies do not do anything by themselves, they are just graphs. The strategy application is what sets the actual traversal in motion, traversing the strategy graph and the input term (see the section on strategy application).

8.3.1 Basic Strategies

```
TTstrat TTbuild_adhoc(TTstrat s, TTrewrite_f r);
TTstrat TTbuild_succeed( void );
TTstrat TTbuild_fail( void );
TTstrat TTbuild_both(TTstrat s1, TTstrat s2);
TTstrat TTbuild_choice(TTstrat s1, TTstrat s2);
TTstrat TTbuild_all(TTstrat s);
TTstrat TTbuild_one(TTstrat s);
```

`TTstrat TTbuild_adhoc(TTstrat s, TTrewrite_f r)`

Lift the rewrite function `r` to the strategy level. If the rewrite function gets applied to a term (when `TTstrat_apply` is called) and fails, the default strategy `s` will be applied instead.

`TTstrat TTbuild_succeed( void )`

This function creates a strategy that, when applied to a term, will return the term itself in the type preserving case and `empty` in the type unifying case.

`TTstrat TTbuild_fail( void )`

This function creates a strategy that will always fail (produce `NULL`) when applied to a term.

`TTstrat TTbuild_both(TTstrat s1, TTstrat s2)`

This function creates a strategy that, when applied to a term, will apply the two strategies in sequence in the type preserving case and concatenate the application of the two strategies to the term in the type unifying case.

`TTstrat TTbuild_choice(TTstrat s1, TTstrat s2)`

This function creates a strategy that, when applied to a term, will try to apply strategy `s1` to a term, if this fails it will apply strategy `s2` to a term.

`TTstrat TTbuild_all(TTstrat s)`

This function creates a strategy that, when applied to a term, will apply `s` to all children of the term, if it fails for one of the children the result will be failure (`NULL`). In the type preserving case a new term will be created with the same function symbol as the old term and the new children. In the type unifying case the new children will be concatenated.

`TTstrat TTbuild_one(TTstrat s)`

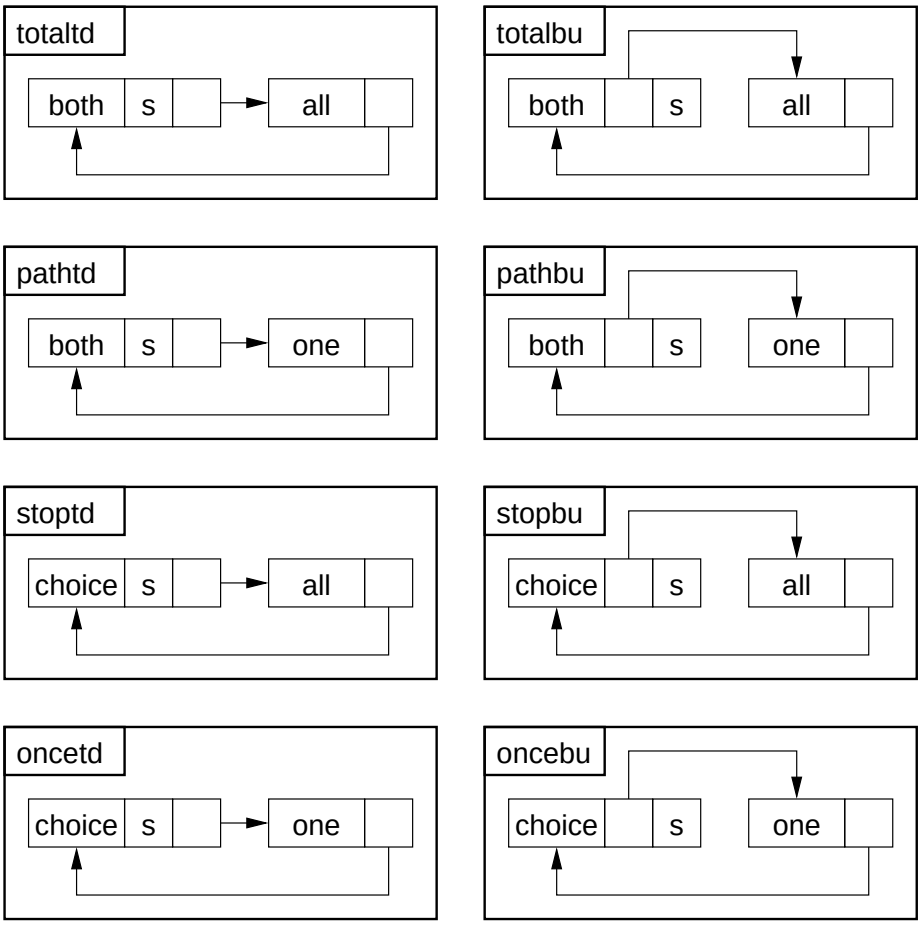
This function creates a strategy that, when applied to a term, will try to apply `s` to one of the children, progressing from left to right. If `s` cannot be applied to any of the children, the result will be failure.

8.3.2 Advanced Strategies

The basic strategies in themselves are not very useful, the real power comes when combining them. Below are various types of traversal ranging from simple top down traversal to more exotic things like stop bottom up.

```
totaltd(s) = both(s, all(this))
totalbu(s) = both(all(this), s)
oncetd(s)  = choice(s, one(this))
oncebu(s)  = choice(one(this), s)
stoptd(s)  = choice(s, all(this))
stopbu(s)  = choice(all(this), s)
```

The `this` stands for the current strategy being defined. To implement such a cyclic structure means that the term library has to support cycles. At the moment it does, but it might be hard to keep supporting cycles when for example reference counting garbage collection is added. The picture below shows what the advanced strategies currently look like internally, i.e. they have cycles.



A way to avoid running into problems with cyclic structures is to have two term libraries: one for normal terms and one for strategic terms. Of course this results in code duplication. A way to avoid using cycles altogether is to mix strategy construction with strategy application, but this is less efficient in both space and time usage. Another issue to consider is that having cycles in a term library is useful for other things, e.g. lazy functional languages allow cyclic terms as well. In short, this is still an open issue.

8.3.3 Strategy Application

Creating strategies is fine, but they don't do anything until they are applied to a term. There are two types of application: type preserving and type unifying. When a basic strategy is applied to a term, two different actions can be taken based on wether the application is type unifying or type preserving. Sometimes the behavior is the same in both cases, e.g. fail. The advanced strategies can be seen as macros, i.e. they are translated to basic strategies.

```
TTterm TTstrat_applytp(TTstrat s, TTterm t);
TTterm TTstrat_applytu(TTstrat s, TTterm t, TTcomb_f comb, TTterm empty);
```

`TTterm TTstrat_applytp(TTstrat s, TTterm t)`

The strategy `s` is applied to the term `t`, the result will either be failure (`NULL`) or a transformed term with the same function symbol as `t`.

```
TTterm TTstrat_applytu(TTstrat s, TTterm t, TTcomb_f comb, TTterm empty)
```

The strategy `s` is applied to the term `t` using `comb` as the concatenation function to combine results (for example list concatenation) and `empty` as the empty result (for example, the empty list).

8.4 Efficient Traversals

Although strategies are very powerful, they are not very efficient. A less powerful, but more efficient way of doing traversals is provided as part of the utility library `ttutils.c`. This section will discuss each traversal and give examples of their use.

```
TTterm TTtransform(TTterm t, TTtrafo_f f)
```

This function performs a type preserving traversal, it applies `f` top down recursively on each term in the tree, if `f` returns `NULL` it will recurse further, otherwise the result of `f` is used to construct the result tree. The example below will expand the string `99` to `9999` in all leafs of the tree.

```
static TTterm
expand_t(TTterm t)
{
    if (TTequal_string(t, "99")) {
        return TTbuild_string("9999");
    }
    return 0;
}

TTterm
expand(TTterm t)
{
    return TTtransform(t, expand_t);
}
```

```
TTterm TTtransform_nonstop(TTterm t, TTrewrite_f f)
```

This function performs a type preserving traversal like `TTtransform`. The difference is that it does not stop, it will first apply the rewrite function to the term `t` and then recursively transform this new term. The example below will perform some kind of simplification on the expressions in an `if` statement. Because `nonstop` is used, nested `if` statements will also be simplified.

```
static TTterm
simplify_t(TTterm t)
{
    TTerm iff, expr, then, stats;

    if (TTmatch_if(t, &iff, &expr, &then, &stats)) {
        return TTbuild_if(iff, simplify(expr), then, stats);
    }
    return t;
}

TTterm
simplify(TTterm t)
{
    return TTtransform_nonstop(t, simplify_t);
}
```

```
TTterm TTanalyze(TTterm t, TTtrafo_f f, TTcomb_f comb, TTterm empty)
```

This function performs a type unifying traversal, it collects information from the parsetree in for example a list or a set.

```
static TTterm
collect_move_a(TTterm t)
{
    if (t->fun == TTfun_move) {
        return TTbuild_singleton(t);
    }
    return 0;
}

TTterm
collect_moves(TTterm t)
{
    return TTanalyze(t, collect_move_a, TTconcat, TTbuild_nil());
}
```

`void TTtraverse(TTterm t, TTtrav_f f)`

This function only does a traversal, the function `f` can return either `FALSE` to continue to recursively traverse the term or `TRUE` to stop, so this is purely for sideffecting traversals, e.g. writing to a file or collecting some result in a static variable. The function `TTtransform` can be used to achieve the same effect and ignoring the constructed result, but `TTtraverse` is more effiicent.

```
static FILE * st_file;

static BOOL
dump_string_t(TTterm t)
{
    const char * str;

    if (TTmatch_string(t, &str)) {
        fprintf(st_file, "%s\n", str);
        return TRUE;
    }
    return FALSE;
}

TTterm
dump_strings(FILE * f, TTterm t)
{
    st_file = f;
    return TTtraverse(t, dump_string_t);
}
```

`void TTtraverse_line(TTterm t, TTtrav_line_f f, int line, int col)`

This function really shows off the power off traversals. Line and column number information is threaded through the traversal. The advantage compared to storing this information in the parsetree is that the parsetree remains lightweight. The disadvantage is that it is less efficient. Decreasing memory consumption seems more important however, especially in the case of COBOL where sources can be anything up to 100K LOC. In case the term library supports maximal sharing, like the ATerm library, the difference in memory consumption becomes even greater: having line and column number information inside the parsetree prevents sharing to a large extend.

```
static void report_move_t(TTterm t, int line, int col)
{
    if (t->fun == TTfun_move) {
        fprintf(stderr, "Move on line %d, col %d.\n", line, col);
    }
}

void report_moves(const char * filename, TTterm t)
{
    TTtraverse_line(t, report_move_t, 1, 1);
}
```

```
TTterm TTselect(TTterm t, TTtrafo_f f)
```

Chapter 9

VS COBOL II

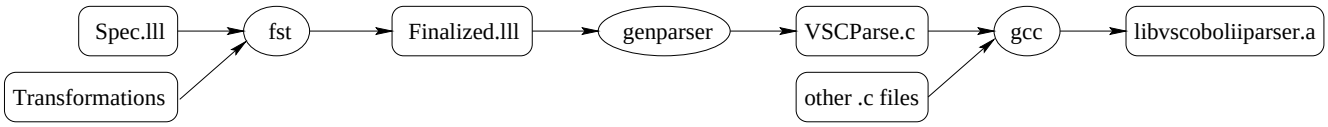
The distribution of GDK comes with the initial VS Cobol II grammar specification as delivered in [3], with all the transformation scripts to further disambiguate and refactor the grammar, and to prepare it for use with different parsing technologies. To illustrate the usefulness of the deployed grammar, a software renovation tool Expand is included in the distribution of GDK. In the present section, we briefly indicate the renovation task addressed by Expand, we describe the process to derive the underlying renovation parser, and finally we sketch the internal structure of Expand. This tool demonstrates how to use a generated renovation parser for a re-engineering task.

A VS Cobol II *program before and after adaptation by* Expand

```
 IDENTIFICATION DIVISION.                IDENTIFICATION DIVISION.
  PROGRAM-ID. LITTLE-Y2K-TEST.            PROGRAM-ID. LITTLE-Y2K-TEST.
 DATA DIVISION.                          DATA DIVISION.
  WORKING-STORAGE SECTION.                WORKING-STORAGE SECTION.
   01  SEEK-NAME       PIC 99.             01  SEEK-NAME       PIC 999.
   01  OTHER-NAME-1    PIC 99.             01  OTHER-NAME-1    PIC 999.
   01  OTHER-NAME-2    PIC 99999.          01  OTHER-NAME-2    PIC 99999.
   01  OTHER-NAME-3    PIC 99.             01  OTHER-NAME-3    PIC 99.
 PROCEDURE DIVISION.                     PROCEDURE DIVISION.
  ...                                     ...
  MOVE SEEK-NAME TO OTHER-NAME-1.         MOVE SEEK-NAME TO OTHER-NAME-1.
  MOVE SEEK-NAME TO OTHER-NAME-2.         MOVE SEEK-NAME TO OTHER-NAME-2.
  IF OTHER-NAME-2 > 99                    IF OTHER-NAME-2 > 299
    ...                                     ...
```

The sample illustrates that we want to expand certain two-digits fields to three digits, that is, `PIC 99` becomes `PIC 999`. This range expansion also triggers the adaptation of literals, namely the replacement of a maximum value `99` by the new value `299`. The affected fields are determined by a seed-and-propagate algorithm. The identification of seed-set elements is based on name heuristics. The propagation relies on a type-of-usage analysis.

Derivation of a Cobol parser



The initial grammar specification `Spec.lll` is transformed by a number of FST scripts resulting in a grammar `Finalized.lll`. This grammar is passed to LLLexport to issue parser generation. The rest of the figure is specific to C-based parsers. The output of LLLexport (maybe after processing by a "parser generator") is compiled by `gcc`. This process is captured in a `Makefile`. To make it straightforward to switch between technologies, all parsers are provided as a library with one top-level function `VSCparse_file` for reading in a file and producing a parse tree.

The *`main` function of the* Expand *tool*

```
#include <stdio.h>
#include "ttterm.h"
```

```
#include "ttutils.h"
#include "parsevsc.h"
#include "expand.h"

int
main(int argc, char **argv)
{
    TTterm pt;

    pt = VSCparse_file("stdin", stdin);
    pt = expand(pt);
    TTunparse(stdout, pt);
    return 0;
}
```

That is, the library function `parseCbl` is invoked resulting in a parse tree `pt`. The renovation task is encoded in the function `expand` which is structured as follows. (i) The seed set is determined. (ii) The propagation is performed. (iii) The picture mask of affected fields is expanded. (iv) Affected literals are adapted. These steps basically amount to traversals which are specific only for a few Cobol patterns. We use generic traversal functionality supplied by TT. The transformed program is finally pretty-printed with `COBPPdump`. The complete C code for the EXPAND tool amounts to 241 LOC.

Bibliography

[1] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of IWPC'98*, pages 108–117, 1998.

[2] R. Lämmel. Grammar Adaptation. In *Proceedings of FME'01*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[3] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.