CS-Note* 76

Andrew J. Cook
May 14, 2004

Fiddling with Variable Names

1 Variables

In the spark vcs there are various entities produced that are represented simply as names. No information about the types of these names is known to CLAM so it assumes that anything it doesn't know about is a variable. However, the vcs may also contain references to constants, record fields and enumerated type values which are not variables.

The information about these values is output into a `.fdl` file. The FDL(Functional Description Language) file contains declarations of types, constants, variables and functions. So if we parse this file into prolog we can pass the information to CLAM for use in proof planning.

2 DCG

To parse the file `.fdl` file we use a DCG(Definite Clause Grammar) in prolog. Because of the limitations of using grammars within prolog the grammar we use seems unnatural in places. The grammar implemented here is primarily meant for the parsing of the automatically generated files from spark and it is unlikely that it is robust enough to handle all the variations possibile in hand coded files.

3 Starting out

Initially the entire contents of the `.fdl` file is read into an array of characters by the `read_file` predicate. And then this is supplied to the grammar predicates to allow them to parse and produce prolog from it.

4 Grammar

4.1 FDL file headers

The `.fdl` file typically consists of a header(which should be ignored) followed by several declarations in between a title statement and an end statement.

To allow the FDL grammar to remove the header text we have:

$$\begin{aligned} fdl &= f\_hd\ fdl \\ &\mid\ space[1]\ fdl \end{aligned}$$

This causes one header line to be removed at a time by allowing the *fdl* to have zero or more headers.

The header lines consist of text within curly brackets ({, }) and so to identify the start of a header line we discard white space and look for an opening curly bracket.

$$\begin{aligned} f\_hd &= \{\ f\_hdc \\ &\mid\ space\ f\_hd \end{aligned}$$

Because the header line can contain any text we parse any text until we come to a closing curly bracket.

$$\begin{aligned} f\_hd &= *[2]\} \\ &\mid\ *\ f\_hdc \end{aligned}$$

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

[1] *space* matches any number(0+) of white spaces.

[2] '*' indicates the matching of any character.

4.2 Declarations

Each line of the `.fdl` file proper is the name of a type of declaration followed by that declaration. So we also have for *fdl*:

$$\begin{array}{rcl} \textit{fdl} & = & \texttt{title}\ \textit{f\_title fdl} \\ & | & \texttt{type}\ \textit{f\_typedec fdl} \\ & | & \texttt{const}\ \textit{f\_const fdl} \\ & | & \texttt{var}\ \textit{f\_var fdl} \\ & | & \texttt{function}\ \textit{f\_func fdl} \end{array}$$

These statements dispatch the various types of declaration to their repective parsers.

After the declarations there is an `end` statement to signify the end of the file.

$$\textit{fdl} \quad = \quad \texttt{end;}\ \textit{space}\ \text{end\_of\_file}$$

4.2.1 *f_title*

The title statement consists of either `function` or `procedure` followed by the name of the sub program that the file was generated from.

$$\begin{array}{rcl} \textit{f\_title} & = & \texttt{function}\ \textit{f\_name}^{3}\texttt{;} \\ & | & \texttt{procedure}\ \textit{f\_name}\texttt{;} \end{array}$$

4.2.2 *f_typedec*

The declaration of a type consists of the name of the type, an equals sign folowed by a type(section 4.3).

$$\textit{f\_typedec} \quad = \quad \textit{f\_name}\ \texttt{=}\ \textit{f\_type}\texttt{;}$$

4.2.3 *f_const* and *f_var*

The declarations of a constants and variables are similar to the declaration of a type. They are simply a name and a colon followed by a type.

$$\begin{array}{rcl} \textit{f\_const} & = & \textit{f\_name}\ \texttt{:}\ \textit{f\_type}\ \texttt{= pending;} \\ \textit{f\_var} & = & \textit{f\_name}\ \texttt{:}\ \textit{f\_type}\texttt{;} \end{array}$$

4.2.4 *f_func*

A function declaration consists of several parts, the name, the argument list and the return type.

$$\textit{f\_func} \quad = \quad \textit{f\_name}\texttt{(}\textit{f\_arglist}\texttt{)}\ \texttt{:}\ \textit{f\_type}\texttt{;}$$

The *f_arglist* is a comma separated list of types for the arguments. We have seperate alternatives for the first/only entry, intermediate entries and a terminating case or for if the arglist is empty.

$$\begin{array}{rcl} \textit{f\_arglist} & = & \textit{space f\_type}\ \texttt{,}\ \textit{f\_arglist} \\ & | & \textit{space f\_type space} \\ & | & \textit{space} \end{array}$$

If the function has the name `round__` then it is ommitted from the output since all of the generated `.fdl` files will contain this standard declaration.

4.3 Types

There are several types which can be present in the `.fdl` file. The simplest, from a parsing point of view, is a built in type like integer or real or a defined type represented by a name. The more complicated compound types are enumerations, records and arrays.

$$\begin{array}{rcl} \textit{f\_type} & = & \textit{f\_name} \\ & | & \texttt{(}\textit{f\_enmlist}\texttt{)} \\ & | & \texttt{record}\ \textit{f\_reclist} \\ & | & \texttt{array[}\textit{f\_arrlist}\texttt{]}\ \texttt{of}\ \textit{f\_type} \end{array}$$

[3] *f_name* simply constucts a name from the start of list of characters by stringing together all characters until it reaches one which isn't eligible(not alphanumeric or an '_').

4.3.1 *f_enmlist*

Enumeration types consist of a list of values that the exist within the type and are very similar to the grammar for arglists. But because there must be at least one entry in the list we do not include the alternative of producing an empty list.

$$\begin{array}{rcl} \textit{f\_enmlist} & = & \textit{space f\_name}\,\texttt{,}\ \textit{f\_enmlist} \\ & | & \textit{space f\_name space} \end{array}$$

4.3.2 *f_reclist*

Record types have a list of named entries along with the type of that entry. The list elements are separated by `;` and the end of the list is signified by `end`.

$$\begin{array}{rcl} \textit{f\_reclist} & = & \textit{space f\_name}\ \texttt{:}\ \textit{f\_type}\,\texttt{;}\ \textit{f\_reclist} \\ & | & \textit{space f\_name}\ \texttt{:}\ \textit{f\_type space}\ \texttt{end} \end{array}$$

4.3.3 *f_arrlist*

An array list is provides syntax for describing n-dimensional arrays it does this by using a list of types so is very similar to the grammar for arglists. Except that, like for enumlists, a zero dimensonal array isn't allowed so the clause for an empty list is ommited.

$$\begin{array}{rcl} \textit{f\_arrlist} & = & \textit{space f\_type}\,\texttt{,}\textit{space f\_arrlist} \\ & | & \textit{space f\_type f\_arrlist} \end{array}$$

5 The Output

The output from the parsing is constructed as the it goes along and returned from the parsing predicates. Each declaration creates a statement which is placed in a list to be returned from the parser.

5.1 Declarations

5.1.1 *f_title*

The `title` statement can take two forms `function` and `procedure` so we use two title wrappers to indicate this.

`title function` *name*`;` ⇒ `fdl_function_name(`*name*`).`
`title procedure` *name*`;` ⇒ `fdl_procedure_name(`*name*`).`

5.1.2 *f_typedec*

Type declarations join the name of the type with its definition.

`type` *name* `=` *type*`;` ⇒ `fdl_type(`*name*`,`*type*`).`

5.1.3 *f_const* and *f_var*

Constants and variables simply join the name to its type.

`const` *name* `:` *type* `= pending;` ⇒ `fdl_constant(`*name*`,`*type*`).`
`var` *name* `:` *type*`;` ⇒ `fdl_variable(`*name*`,`*type*`).`

5.1.4 *f_func*

Function declarations require a name, a list of argument types and an output type.

`function` *name*`(`*arglist*`) :` *type*`;` ⇒ `fdl_function(`*name*`,`*arglist*`,`*type*`).`

The *arglist* is a list of types generated by *f_arglist*(section 4.2.4).

5.2 Types

The type statements are generated in the same way as for the declarations except that the prolog will be use within declarations instead of inserted into a list.

5.2.1 Array

$$\texttt{array[}\mathit{itype}\texttt{] of }\mathit{etype} \Rightarrow \texttt{array(}\mathit{arrlist}\texttt{,}\mathit{type}\texttt{)}$$

5.2.2 Enumerated

$$\texttt{(}\mathit{enumlist}\texttt{)} \Rightarrow \texttt{enum(}\mathit{enumlist}\texttt{)}$$

enumlist is a list of types created by *f_enmlist*(section 4.3.1).

5.2.3 Record

$$\texttt{record }\mathit{reclist}\texttt{ end} \Rightarrow \texttt{record(}\mathit{reclist}\texttt{)}$$

reclist is a list of record elements created by *f_reclist*(section 4.3.2). Each record element consists of the field name and a type.

$$\mathit{name}\texttt{:}\mathit{type} \Rightarrow \texttt{rel(}\mathit{name}\texttt{,}\mathit{type}\texttt{)}$$

6 Conclusion

The grammar presented here is not entirely general but it should be sufficient to parse the machine generated `.fdl` files produced by the spark examiner.

A Table of transformations

Declarations:	
`title function` *name*`;`	⇒ `fdl_function_name(`*name*`).`
`title procedure` *name*`;`	⇒ `fdl_procedure_name(`*name*`).`
`type` *name* `=` *type*`;`	⇒ `fdl_type(`*name*`,`*type*`).`
`const` *name* `:` *type* `= pending;`	⇒ `fdl_constant(`*name*`,`*type*`).`
`var` *name* `:` *type*`;`	⇒ `fdl_variable(`*name*`,`*type*`).`
`function` *name*`(`*arg1*`,...,`*argn*`) :` *type*`;`	⇒ `fdl_function(`*name*`,[`*arg1*`,...,`*argn*`],`*type*`).`
Types:	
`array [`*t1*`,...,`*tn*`] of` *etype*	⇒ `array([`*t1*`,...,`*tn*`],`*etype*`)`
`(`*e1*`,...,`*en*`)`	⇒ `enum([`*e1*`,...,`*en*`])`
`record` *r1*`,...,`*rn* `end;`	⇒ `record([`*r1*`,...,`*rn*`])`

B Example

The following is an artificially constrtucted example of an `.fdl` file made by pasting examples of the the various constructs from other files into the `filters.fdl` file.

```
          {*******************************************************}
                                   {FDL Declarations}
       {SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01}
                              {Demonstration Version}
          {*******************************************************}


                          {DATE : 21-APR-2004 16:00:15.62}

                          {procedure Filter_Sum.Filter_S}


title procedure filter_s;

  function round__(real) : integer;
  type d_type = array [integer] of integer;
  type array_type = array [integer] of integer;
  type table_type = record
        size : integer;
        elem : array_type
     end;
  type enumerated_index = (literal_a, literal_b, literal_c,
     literal_d, literal_e);
  const i_type__last : integer = pending;
  const i_type__first : integer = pending;
  const integer__last : integer = pending;
  const integer__first : integer = pending;
  var loop__1__i : integer;
  var r : integer;
  var d : d_type;
  var table : table_type;
  function round(real) : integer;
  function minimum(array_type, integer, integer) : boolean;

end;
```

The result of parsing this file is:

```
fdl_procedure_name(filter_s).
fdl_type(d_type,array(integer,integer)).
fdl_type(array_type,array(integer,integer)).
```

```
fdl_type(table_type,record([rel(size,integer),rel(elem,array_type)])).
fdl_type(enumerated_index,enum([literal_a,literal_b,literal_c,literal_d,literal_e])).
fdl_constant(i_type__last,integer).
fdl_constant(i_type__first,integer).
fdl_constant(integer__last,integer).
fdl_constant(integer__first,integer).
fdl_variable(loop__1__i,integer).
fdl_variable(r,integer).
fdl_variable(d,d_type).
fdl_variable(table,table_type).
fdl_function(round,[real],integer).
fdl_function(minimum,[array_type,integer,integer],boolean).
```

C The prolog

The prolog code for the parsing is as follows:

```
:- include('characters.pl').

show_file(File):-
    open(File, read, Sors),
    process_file(user_output, Sors),
    close(Sors),
    !.
show_file(Orig, Outf):-
    open(Orig, read, Sors),
    open(Outf, write, Dest),
    process_file(Dest, Sors),
    close(Sors),
    close(Dest),
    !.

process_file(Dest, Sors):-
    read_file(Sors, Fill),
    parser(Parsed, Fill),
    show_list(Dest, Parsed),
    !.

show_list(Dest, [H|T]):-
    write(Dest, H),write(Dest, '.'),nl(Dest),
    show_list(Dest, T),
    !.
show_list(_, []):-
    !.

/*******************************************************
*read_file: reads the whole file into a list of characters*
*******************************************************/
read_file(Sors, List):- read_file(Sors, 'A', List).
read_file(Sors, -1, []):- close(Sors).%Not needed but in here just in case
read_file(_, end_of_file, []).
read_file(Sors, _, List):-
    get_char(Sors, L),
    read_file(Sors, L, Ist),
    List = [L|Ist],
    !.

/*************************************************
*parser: parses the character list according to the *
```

```
*fdl file grammar and returns a list of predicates  *
*corresponding to the statements in the list.       *
****************************************************/
parser(Parsed, List):-
    phrase(fdl(Parsed),List),
    !.

/****************************************************
*fdl > strip off headers.                           *
*    | end of file is reached.                      *
*    | strip ' ' and '\n'.                          *
*    | the useful part of the document.             *
****************************************************/
fdl(Fdl) --> f_hd, fdl(Fdl).
fdl([]) --> [end_of_file].
fdl(Fdl) --> " ", fdl(Fdl).
fdl(Fdl) --> "\n", fdl(Fdl).
fdl([Fdl_t|Fdl_r]) --> "title ", f_title(Fdl_t), fdl(Fdl_r).
fdl([Fdl_y|Fdl_r]) --> "type ", f_typedec(Fdl_y), fdl(Fdl_r).
fdl([Fdl_c|Fdl_r]) --> "const ", f_const(Fdl_c), fdl(Fdl_r).
fdl([Fdl_v|Fdl_r]) --> "var ", f_var(Fdl_v), fdl(Fdl_r).
fdl([Fdl_f|Fdl_r]) --> "function ", f_func(Fdl_f), fdl(Fdl_r).
fdl(Fdl_r) --> "function ", f_func, fdl(Fdl_r).
fdl(R) --> "end;", spc, fdl(R).
%fdl(T) --> [L], {write(L),nl}, fdl(T);

/****************************************************
*spc > remove whitespace.Not used at present        *
****************************************************/
spc --> " ",!, spc.
spc --> "\n",!, spc.
spc --> "\t",!, spc.
spc --> "".
/****************************************************
*f_hd > a '{' and the rest of the line.             *
*      | strip a space or new line.                 *
****************************************************/
f_hd --> "{", f_hdc.
f_hd --> " ", f_hd.
f_hd --> "\n", f_hd.
/****************************************************
*f_hdc > strip characters until a '}' is            *
*              encountered.                         *
****************************************************/
f_hdc --> [_], "}".
f_hdc --> [_], f_hdc.

/****************************************************
*f_title > procedure _name_;                        *
*       | function _name_;                          *
****************************************************/
f_title(fdl_procedure_name(Name)) -->
    "procedure ",!,
    f_name(Name),!,
    ";",
    !.
f_title(fdl_function_name(Name)) -->
    "function ",!,
    f_name(Name),!,
```

```
    ";",
    !.
/****************************************************
*f_typedec > _name_ = _type_;                       *
****************************************************/
f_typedec(fdl_type(Name,Type)) -->
    f_name(Name),!,
    " = ",
    f_type(Type),
    ";",
    !.
/****************************************************
*f_const > _name_ : _type_ = pending;               *
****************************************************/
f_const(fdl_constant(Name,Type)) -->
    f_name(Name),
    " : ",
    f_type(Type),
    " = pending",
    ";",
    !.
/****************************************************
*f_var > _name_ : _type_;                           *
****************************************************/
f_var(fdl_variable(Name,Type)) -->
    f_name(Name),
    " : ",
    f_type(Type),
    ";",
    !.
/****************************************************
*f_func > _name_(_arglist_) : _type_;               *
****************************************************/
f_func(fdl_function(Name,Args,Type)) -->
    f_name(Name),
    {Name \= 'round__'},
    "(",
    f_arglist(Args),
    ")",
    " : ",
    f_type(Type),
    ";",
    !.
f_func -->
    "round__(real) : integer;",
    !.
/****************************************************
*f_arglist > _type_, _arglist_                      *
*           | _type_                                *
****************************************************/
f_arglist([Arg|List]) -->
    spc,
    f_type(Arg),
    ",",
    f_arglist(List),
    !.
f_arglist([Arg|[]]) -->
    spc,
    f_type(Arg),
```

```
    spc,
    !.
f_arglist([]) --> "".

/*****************************************************
*f_type > array [_type_] of _type_                  *
*       | record _reclist_                          *
*       | (_arglist_)                               *
*       | integer                                   *
*****************************************************/
f_type(array(Index,Value)) -->
    "array ",
    "[",
    f_arrlist(Index),
    "]",
    " of ",
    f_type(Value),
    !.
f_type(record(List)) -->
    "record",
    f_reclist(List),
    !.
f_type(enum(Args)) -->
    "(",
    f_enmlist(Args),
    ")",
    !.
f_type(Name) -->
    f_name(Name),
    !.
/*****************************************************
*f_arrlist > _type_, _arrlist_                      *
*          | _type_                                 *
*****************************************************/
f_arrlist([Type|List]) -->
    spc,
    f_type(Type),
    ",",
    f_arrlist(List),
    !.
f_arrlist([Type|[]]) -->
    spc,
    f_type(Type),
    spc,
    !.

/*****************************************************
*f_reclist > _name_ : _type_; _reclist_             *
*          | _name_ : _type_ end                    *
*****************************************************/
f_reclist([rel(Name,Type)|List]) -->
    spc,
    f_name(Name),
    " : ",
    f_type(Type),
    ";",
    f_reclist(List),
    !.
f_reclist([rel(Name,Type)|[]]) -->
```

```
    spc,
    f_name(Name),
    " : ",
    f_type(Type),
    spc,
    "end",
    !.

/****************************************************
*f_enmlist > _name_, _enumlist_                    *
*          | _name_                                *
****************************************************/
f_enmlist([Arg|List]) -->
    spc,
    f_name(Arg),
    ",",
    f_enmlist(List),
    !.
f_enmlist([Arg]) -->
    spc,
    f_name(Arg),
    spc,
    !.

/****************************************************
*f_name > constructs a name.                       *
****************************************************/
f_name(Name) -->        %First letter
    {alphaNumericAda(N)},
    [N],
    f_namer(Aml),
    {atom_chars(Name, [N|Aml])}.
f_namer([N|Ame]) -->    %Rest letters
    {alphaNumericAda(N)},
    [N],
    f_namer(Ame),
    !.
f_namer([]) --> "".     %No more letters
```