SPRUT (internal representation description translator)

Vladimir Makarov, vmakarov@gcc.gnu.org Apr 5, 2001

This document describes SPRUT (translator of a compiler internal representation into standard procedural interface).

Contents

1 Introduction **1**

2 Internal representation description language **1**

2.1 Layout of internal representation description 2

2.2 Declarations 2

2.3 Types of nodes 3

3 Standard Procedural Interface **6**

3.1 C SPI 6

3.1.1 C SPI types, variables, macros 6

3.1.2 C Functions 7

3.2 C++ SPI 13

3.2.1 C++ SPI types, variables, macros 13

3.2.2 C++ Functions 14

3.3 Type specific macros 19

3.4 Storage management macros 21

4 SPRUT Usage **21**

5 Appendix 1 - Syntax of internal representation description language **25**

1 Introduction

SPRUT is a translator of a compiler internal representation description (IRD) into Standard Procedural Interface (SPI). The most convenient form of the internal representation is a directed graph. IRD defines structure of the graph. SPI provides general graph manipulating functions. The defined graph nodes can be decorated with attributes of arbitrary types.

IRD declares types of nodes of the graph. Nodes contains fields, part of them represents links between nodes, and another part of them stores attributes of arbitrary types. To make easy describing internal representation the IRD supports explicitly single inheritance in node types and also can model multiple inheritance. There

can be several levels of internal representation description in separate files. The nodes of one level refer to the nodes of previous levels. Therefore each next level enriches source program internal representation. For example, the zero level representation may be internal representation for scanner, the first level may be internal representation for parser, and so on.

SPI can contains functions to construct and destroy graphs and graph nodes, to copy graphs or graph nodes, to read and write graphs or graph nodes from (to) files, to print graphs or graph nodes, to check up constraints on graph, to traverse graphs, and to transform acyclic graphs in some commonly used manners. SPI can also check up the most important constraints on internal representation during work with node fields. SPI can automatically maintain back links between internal representation nodes.

Using SPRUT has the following advantages:

1. brief and concise notation for internal representation
2. improving maintainability and as consequence reliability of the compiler
3. user is freed from the task of writing large amounts of relatively simple code

The following sections define the internal representation description language, explain SPI, and present some examples.

2 Internal representation description language

IRD declares types of nodes of the graph. Nodes contains fields, part of them represents links between nodes, and another part of them stores attributes of arbitrary types. To make easy describing internal representation the IRD supports explicitly single inheritance in node types and also can model multiple inheritance. There can be several levels of internal representation description in separate files. The nodes of one level refer to the nodes of previous levels. Therefore each next level enriches source program internal representation.

2.1 Layout of internal representation description

To describe internal representation a special language is used. An internal representation description structure has the following layout which is similar to one of YACC file.

```
DECLARATIONS
%%
TYPES OF NODES
%%
ADDITIONAL C/C++ CODE
```

The '%%' serves to separate the sections of description. All sections are optional. The first '%%' starts section of description of types of internal representation nodes and is obligatory even if the section is empty, the second '%%' may be absent if section of additional C/C++ code is absent too.

The section of declarations may contain names of predefined types of fields of internal representation nodes and names of types of double linked nodes. The section also contains name of original internal representation description if given file contains extension of an internal representation description. And finally the section may contains sections of code on C/C++.

The next section contains description of types of internal representation nodes.

The additional C/C++ code can contain any C/C++ code you want to use. Often functions which are not generated by the translator but are needed to work with internal representation go here. This code without changes is placed at the end of file generated by the translator.

2.2 Declarations

The section of declarations may contain the following construction.

```
%type IDENTIFIER ...
```

All predefined types must be defined in constructions of such kind. The same name can be defined repeatedly. All references to node whose type is a sub-type of type with name present in the following construction will be double linked.

```
%double IDENTIFIER ...
```

It means that SPI will generates functions (macros) which permit to examine all fields (may be in other nodes) described as of given node type (or its subtype) which refer to node of given type (or its sub-type), i.e. fields which are described as of given node type (or its subtype) will be double linked (see below). SPI will automatically maintain such double links. The simplest way to describe double linked graph is to insert construction

```
%double %root
```

Last construction in the section of declarations of kind

```
%extend IDENTIFIER
```

defines name (without suffix) of file containing source internal representation which is extended by given file. The file contains original internal representation if there is no one such construction. The original specification file has level 0, the extensions have level 1, 2, and so on. This feature permits sequentially to develop an internal representation and to save and restore any its level (see SPI) to additional tools, e.g. browsers. For example, there may be three levels of source program internal representation. The zero level representation may be internal representation for semantic analysis, the first level may be a low level machine-dependent internal representation, and the second level may be used to generate object code. Only first such construction in one file is essential. All subsequent such constructions are ignored.

There may be also the following constructions in the declaration section

```
%local {
   C/C++ DECLARATIONS
}

%import {
   C/C++ DECLARATION
}
```

```
    and

%export {
   C/C++ DECLARATION
}
```

which contain any C/C++ declarations (types, variables, macros, and so on) used in the description sections.

The local C/C++ declarations are inserted at the begin of generated implementation file (see SPI description) but after include-directive of interface file.

C/C++ declarations which start with '%import' are inserted at the begin of generated interface file. For example, such C/C++ code may contain C/C++ definitions of predefined types which are used in field declarations of node types.

C/C++ declarations which start with '%export' are inserted at the end of generated interface file. For example, such C/C++ code may contain definitions of external variables and functions which refer to node type representation (see type 'IR_node_t' in SPI description).

All C/C++ declarations can redefine all type specific and internal macros (see SPI) because they are placed in implementation file. All C/C++ declarations are placed in the same order as in the section of declarations. C/C++ declarations from IRD file with smaller level number (see construction '%extend') are placed in interface or implementation files firstly.

2.3 Types of nodes

The section of declarations is followed by section defining internal representation node types. An internal node type is described by the following construction

```
%abstract
IDENTIFIER :: IDENTIFIER (or %root)
CLASS FIELDS
SKELETON FIELDS
OTHER FIELDS
```

Keywords '%abstract' is optional. Node type description which starts with this keyword denotes abstract node type, i.e. node of such type does not exist in internal representation of any source program. Abstract nodes types serve only to description of common fields of several node types.

The first identifier defines name of internal representation node type, the second defines name of node type all declarations of fields of which are inherited into the given node type. In this construction the first node type is so called immediate super type, the second is immediate sub-type.

Node type A is a super-type of node type B (and node type B is a sub-type of node type A) iff node type A is immediate super-type of super-type of node type B. All node types are sub-types of implicitly declared node type with name '%root'. There are also nodes of special type (error nodes). Type of these nodes are believed to be sub-type of all declared node types. The definition of type of error nodes are absent in any internal representation description.

The identifier of immediate super-type (with '::') can be absent. In this case the construction is continuation of given type node declaration. There can be only the single main node type declaration and many

its continuations. The order of main node type declaration and its continuations can be arbitrary. The continuations can not start with keyword '%abstract'.

Construction of the following kind

```
A, B, ... :: C
DECLARATIONS OF FIELDS
```

is abbreviation of the following constructions

```
A :: C
DECLARATIONS OF FIELDS
B :: C
DECLARATIONS OF FIELDS
...
```

The fields are sub-divided on kinds. There are three kinds of fields.

1. Class fields. There is the single instance of a class field for all nodes of given type.
2. Skeleton fields. There is the single instance of a skeleton fields for each node of given type. The value of skeleton field is to be given by user at the node creation moment (see SPI description).
3. Other fields. This kind of fields is analogous to one of skeleton fields but value of such field is not given by user at the node creation moment.

Optional sections of declarations of class fields, skeleton fields, and other fields start correspondingly with keywords '%class', '%skeleton', and '%other'.

```
%class LIST OF DECLARATIONS OF FIELDS

%skeleton LIST OF DECLARATIONS OF FIELDS

%other LIST OF DECLARATIONS OF FIELDS
```

These keywords are followed by may be empty list of declarations of fields. The list elements are field declaration or target code. Field declaration is described by the following construction:

```
IDENTIFIER : FIELD  %double  TYPE  CONSTRAINTS  ACTIONS
```

Identifier is name of given field. All declarations of fields of a node type must have unique names. But declarations of fields of different node types can have the same names if the types of such fields are the same, the fields are simultaneously described as double linked or not, and all such fields are class or any non-class fields. Owing to the feature it is possible to model multiple inheritance. Semicolon ':' is followed by the field type. There are the following types of nodes fields:

1. identifier from a clause '%type'. The identifier represents predefined type and must be declared anywhere by the construction 'typedef' of C/C++.
2. node name. This type represents arc to a node of given type or its subtypes and is implemented by pointer of C/C++.

These constructions can have optional clause '%double'. Its sense is slightly different from the one in the declarations section. This clause means that SPI will generate functions (macros) which permits to examine all nodes of the declared type (or its sub-type) which refer through given field to a node of the field type (or its sub-type). The clause '%double' can not be given for class fields.

The field type may be followed by constraints and actions in any order. The constraint is usually present for non-null value node reference. The constraint is a boolean expression on C/C++ in brackets '[' and ']'. The constraints are tested with the aid of some generated functions in the same order as they are present in corresponding type node declaration. The actions can contain any statements on C/C++ in figure brackets '{' and '}' (the brackets are also output therefore C/C++ declarations can be in actions). The actions for skeleton and other fields are fulfilled at the node creation time in the same order as they are present in corresponding type node declaration. The actions for class fields are fulfilled at the internal representation initiation time.

Constructions '$$', '$' in the constraints and the actions represent correspondingly current node and previous field. But '$' is not changed by the previous field if the previous field in given node type declaration is of other kind than the constraint or action (e.g. the field of skeleton kind and the constraint of class kind) or such field does not exist in the current node type declaration. Also it should be remembered that '$' can not be used in left hand side of assignment if the construction represent double linked fields.

The construction 'IDENTIFIER : FIELD TYPE' may be absent. This case is convenient for definition of additional constraints and actions for fields which declared in a super-type of given node type.

Construction of kind

```
A, B, ... : C ...
```

is abbreviation of the following constructions

```
A : C ...
B : C ...
```

Full YACC syntax of internal representation description language is placed in Appendix 1.

3 Standard Procedural Interface

An internal representation description is translated by SPRUT (internal representation definition translator) into a standard procedural interface (SPI). If the specification file is an extension (see the language description) of another specification file the later is also used to SPI generation and so on. SPI consists of interface and implementation files having the same names as one of the specification file and correspondingly suffixes '.h' and '.c' (C code) or '.cpp' (C++ code).

3.1 C SPI

By default (when option '-c++' is not present on the command line) SPRUT generates C SPI. The following subsections describe C SPI.

3.1.1 C SPI types, variables, macros

Interface file of SPI consists of C declarations of the following types, variables and macros.

1. Type 'IR_node_mode_t' is represented by enumeration definition. There is enumeration constant with node type name and prefix 'IR_NM_' for each node type declaration even for abstract node type. Also there are enumeration constants with names 'IR_NM__root' and 'IR_NM__error' for and predefined type '%root' and for error nodes. 'IR_NM' is abbreviation of internal representation node mode. There is also the enumeration name 'IR_node_mode_enum' which is convenient to use in imported section for forward definition of pointer to this enumeration.

2. Type 'IR_node_t' represents a node (more correctly pointer to a structure representing node). There are the type value 'NULL' reserved to represent none node. The structure of this type is opaque and all work with nodes are to be fulfilled through macros (or/and functions). But to show efficacy of types implementation it should be said that every node type is turned into a structure declaration. All these structures have first member of type 'IR_node_mode_t'. Other members represent the node fields except for class field. Depending on SPRUT command line option '-flat-structure' (see SPRUT usage) this structure can contain also members corresponding to fields declared in super types of given node type or member which type is structure corresponding to immediate super type of given node type. Class fields for each non-abstract node type are represented by only one separate structure.

   The fields in different node types with the same name (when it is consequence of existing common super type with this field) have the same displacement from the structure begin because types of previous fields is guaranteed to be the same in these structures. The fields in different node types with the same name (when it is not consequence of existing common super type with this field) may have the different displacement from the structure begin therefore access to such fields requires additional indirect addressing.

   Double linked field (see internal representation description language) requires more memory (about in five times) for its implementation. Speed of SPI work with double linked fields is sightly slower. Because additional work is needed only to setting up the field value and my experience shows that usually number of setting up fields value is less than number of their fetching about in ten times.

   Type 'IR_node_t' is simply pointer to struct representing node type '%root'. The name of this structure is 'IR_node'. This name can be used in imported section for forward reference for nodes (instead of 'IR_node_t').

3. Optional variable 'IR_node_name' is array mapping values of type 'IR_node_mode_t' to names (strings) in C. (see also option -no-node-name in SPRUT usage).

4. Variable 'IR_node_size' is array mapping values of type 'IR_node_mode_t' to size of corresponding structures on C.

5. Macro 'IR_IS_TYPE(type, supertype)' is evaluated as nonzero if the first argument of type 'IR_node_mode_t' is equal to the second argument or node type represented by the first argument is a sub-type of node type represented by the second argument. Otherwise the macro returns zero. But this macro returns always zero for any error node, i.e. this macro can be used to define that the node of the first mode has fields of the node of the second mode.

6. Macro 'IR_NODE_MODE(node)' returns node mode represented by value of type 'IR_node_mode_t' for node given as the macro argument of type 'IR_node_t'.

7. Macro ‘IR_IS_OF_TYPE(node, supertype)’ is abbreviation of frequently used ‘IR_IS_TYPE (IR_NODE_MODE (node), supertype)’.

8. Macro ‘IR_NODE_LEVEL(node)’ has value which is internal representation description level in which declaration of given node type starts.

3.1.2 C Functions

SPI interface file also contains definitions of generated functions to creation, deletion, modification, comparison, copy, check, print, and binary input/output of internal representation nodes and functions for access to values of internal representation node fields.

These functions do not permit to work with abstract nodes. Only one graph function can be active, e.g. call of function ‘IR_check_graph’ is not permitted during work of any traverse or transformation function.

Macros can be generated with or instead of the functions for access to internal representation node fields. This feature should be used if used C preprocessor has not rigid constraints on number of defined macros. Because macros arguments may be evaluated many times no side-effects should be in the arguments.

1. Access functions (or/and macros). These functions are generated when option ‘-access’ is present in the SPRUT command line. If option ‘-macro’ is used the access functions and macros are generated. If option ‘-only-macro’ is used only access macros are generated. Only option ‘-macro’ or ‘-only-macro’ can be on the command line of SPRUT. There is one function for each field with unique name. Access function has name formed from prefix ‘IR_’ and corresponding field name. There is one exception. When access functions and macros are generated, macros have names starting with prefix ‘IR_’ and the functions have names starting with prefix ‘IR_F_’. Access function has one argument of type ‘IR_node_t’. The argument represents node which must have such field. Type of returned value is predefined type (if the field is of such type) or ‘IR_node_t’ (if the field is of a node type). Functions

```
‘IR_double_link_t IR__first_double_link (IR_node_t node)’,

‘IR_double_link_t IR__next_double_link
       (IR_double_link_t prev_double_link)’,

‘IR_double_link_t IR__previous_double_link
       (IR_double_link_t next_double_link)’
```

   return references to fields which have double links to given node. Functions ‘IR__first_double_link’ is always implemented by function. The node itself which contains such links can be determined by function

```
‘IR_node_t IR__owner (IR_double_link_t link)’.
```

   Here type ‘IR_double_link_t’ represents pointer to double fields. The functions may also check up that the node has given field (see SPRUT options).

2. Modification functions. These functions are generated when option ‘-set’ is present in the SPRUT command line. There is one functions for each field with unique name. Modification function has name formed from prefix ‘IR_set_’ and corresponding field name. Modification function has two arguments. The first argument of type ‘IR_node_t’ represents node which must have such field. The second argument of predefined type (if the field is of such type) or of type ‘IR_node_t’ (if the field is of a node type) represents new value of the field.

The function may check up constraint if the field represents arc to node of a node type or its sub-types (see SPRUT options). The functions may also check up that the node has given field.

There is also function for modification of double links

```
'void IR__set_double_link (IR_double_link_t double_link,
                           IR_node_t value)'
```

Remember that values of 'IR__next_double_link' and 'IR__previous_double_link' are changed after calling this function.

3. Function

   ```
   'IR_node_t IR_create_node (IR_node_mode_t node_mode)'
   ```

   is generated in any case. This function allocates storage for node of any type (except for abstract) represented by the argument (it can be also 'IRNM__error), sets up node mode and node level, fulfills fields assignments (except for class fields) in the same order as the fields are declared. The field assignment is fulfilled according to type specific operation (see section 'Type specific macros') and actions which are defined after given field in sections of skeleton and other fields of corresponding node type. After that the function returns the created node. The single way to create error nodes is to use this function with argument 'IRNM__error'.

4. Node type specific creation functions. These functions are generated when option '-new' is present in the SPRUT command line. There is one function for each node type (except for abstract). Names of functions are formed from prefix 'IR_new_' and corresponding node type name. The number of parameters of a function is equal to one of skeleton fields of corresponding node type. The parameters of these functions have to be given in the order of the skeleton fields in the specification. Fields of the super-type (recursively) precede the ones of the sub-type. These functions may check up constraints on skeleton fields which represent arc to node of a node type or its sub-types (see SPRUT options). These functions call node creation function and after that assign parameters values to corresponding skeleton fields. After that the functions return the created node. These functions provide a mechanism similar to record aggregates. Nested calls of such functions allow programming with terms as in LISP.

5. Function

   ```
   'void IR_free_node (IR_node_t node)'
   ```

   is generated when option '-free' or '-free-graph' is present in the SPRUT command line. This function deletes given node. The deletion consists of evaluation of finalization macros for node fields (see field type specific macros) in reverse order and all node space deallocation. The function also delete double linked fields of given node from the corresponding lists of double linked fields. But the user should know that there may be dangling references to given node itself after the function call. The function does nothing if the parameter value is equal to NULL.

6. Function

   ```
   'void IR_free_graph (IR_node_t graph)'
   ```

   is generated when option '-free-graph' is present in the SPRUT command line. This function deletes given node and all nodes accessible from given node 'graph'. The nodes are deleted in depth first order. The function does nothing if the parameter value is equal to NULL.

7. Function

```
'void IR_conditional_free_graph
         (IR_node_t graph,
          int (*guard) (IR_node_t node))'
```

is generated when option '-free-graph' is present in the SPRUT command line. This function deletes given node and all nodes accessible from given node 'graph'. The nodes are deleted in depth first order. The function does nothing if the parameter value is equal to NULL. The processing (traversing and deleting) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All graph is deleted when the returned value is always nonzero.

8. Function

```
'IR_node_t IR_copy_node (IR_node_t node)'
```

is generated when option '-copy' or '-copy-graph' is present in the SPRUT command line. This function makes copy of given node. Field type specific macros are used (see section 'Type specific macros') to make this. The function only returns NULL if the parameter value is equal to NULL.

9. Function

```
'IR_node_t IR_copy_graph (IR_node_t graph)'
```

is generated when option '-copy-graph' is present in the command line. This function makes copy of graph given by its node 'graph'. The function only returns NULL if the parameter value is equal to NULL.

10. Function

```
'IR_node_t IR_conditional_copy_graph
             (IR_node_t graph,
              int (*guard) (IR_node_t node))'
```

is generated when option '-copy-graph' is present in the command line. This function makes copy of graph given by its node 'graph'. The function only returns NULL if the parameter value is equal to NULL. The processing (traversing and copying) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All nodes of graph are copied when the returned value is always nonzero.

11. Function

```
'int IR_is_equal_node (IR_node_t node_1,
                       IR_node_t node_2)'.
```

is generated when option '-equal' or '-equal-graph' is present in the SPRUT command line. This function returns nonzero iff 'node_1' is equal to 'node_2', i.e. node_1 and node_2 have the same type and all fields of 'node_1' are equal to the corresponding fields of 'node_2'. Field type specific macros are used (see section 'Type specific macros') to make this. The function return nonzero if parameter values are equal to NULL.

12. Function

```
'int IR_is_equal_graph (IR_node_t graph_1,
                        IR_node_t graph_2)'
```

is generated when option '-equal-graph' is present in the SPRUT command line. This function returns nonzero iff graph starting with node 'graph_1' is equal to graph starting with node 'graph_2', i.e. the graphs have the same structure and all predefined type fields of 'graph_1' are equal to the corresponding fields of 'graph_2'. The function returns TRUE if parameter values are equal to NULL.

13. Function

```
'int IR_is_conditional_equal_graph
       (IR_node_t graph_1, IR_node_t graph_2,
        int (*guard) (IR_node_t node))'
```

is generated when option '-equal-graph' is present in the SPRUT command line. This function returns nonzero iff graph starting with node 'graph_1' is equal to graph starting with node 'graph_2', i.e. the graphs have the same structure and all predefined type fields of 'graph_1' are equal to the corresponding fields of 'graph_2'. The function returns TRUE if parameter values are equal to NULL. The processing (traversing and comparing) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All nodes of the graphs are compared when the returned value is always nonzero. Unprocessed graph nodes are believed to be equal.

14. Function

```
'int IR_check_node (IR_node_t node,
                    int class_field_flag)'
```

This function is generated when option '-check' or '-check-graph' is present in the SPRUT command line. By default the most of generated functions (and macros) which work with internal representation nodes do not check up constraints which described in the specification. There are two kinds of constraints. The first is constraint on field which represents arc to node of given type or its sub-types. The second is constraint which defined by target code (see the internal representation definition language). The user is responsible to adhere to the constraints. Function 'IR_check_node' can be used to check all constraints corresponding to given node. The function does nothing if the parameter value is equal to null. In case of a constraint violation the involved node and all child nodes are printed on standard error and the function returns nonzero (error flag). If the second parameter value is zero then class fields of given node are not checked up.

15. Function

```
'int IR_check_graph (IR_node_t graph,
                     int class_field_flag)'
```

is generated when option '-check-graph' is present in the sprut command line. This function checks given node 'graph' and all nodes accessible from given node. It should be known that a class field is processed only when the second parameter value is nonzero. The function does nothing if the parameter value is equal to null. In case of fixing any constraint violation returns nonzero (error flag).

16. Function

```
'int IR_conditional_check_graph
       (IR_node_t graph, int class_field_flag,
        int (*guard) (IR_node_t node))'
```

is generated when option '-check-graph' is present in the sprut command line. This function checks given node 'graph' and all nodes accessible from given node. It should be known that a class field

is processed only when the second parameter value is nonzero. The function does nothing if the parameter value is equal to null. In case of fixing any constraint violation returns nonzero (error flag). The processing (traversing and checking) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All nodes of graph are checked when the returned value is always nonzero. It is better to free changed nodes which are not used after transformation especially when there are double links to the nodes (otherwise the nodes can be accessible with the aid of functions for work with double links).

17. Function

```
'void IR_print_node (IR_node_t node,
                     int class_field_flag)'
```

is generated when option '-print' is present in the sprut command line or options '-check' or 'check-graph' is present because check functions use this print function. The function outputs values of node fields with their names to standard output in readable form. To make this the function outputs sequentially all the node fields (including class fields if the second parameter value is nonzero) with the aid of field type specific macros (see below). The function does nothing if the parameter value is equal to null.

18. Functions

```
'int IR_output_node (FILE *output_file, IR_node_t node,
                     int level)',

'IR_node_t IR_input_node (FILE *input_file,
                          IR_node_t *original_address)'
```

are generated when option correspondingly '-output' or '-input' is present in the SPRUT command line. Function 'IR_output_node' writes node to given file. To make this the function outputs the second parameter value, node address and after that sequentially all the node fields (except for class fields) whose declarations are present in internal representation description of 'level' or less than the one (see internal representation description language). The node type must be also declared in internal representation description of 'level' or less than the one. The function does nothing if the value of parameter 'node' is equal to NULL. A nonzero function result code means fixing an error during output.

Function 'IR_input_node' reads node from given file, allocates space for the node, initiates its fields and changes the node fields output early. To make this the function uses function 'create_node'. The output level of input node must be less or equal to the internal representation level of given SPI. Null value returned by the function means fixing an error during input. The function returns original address of input node through the second parameter. The output and input of fields is fulfilled with the aid of field type specific macros (see section 'Type specific macros').

19. Function

```
'void IR_traverse_depth_first
      (IR_node_t graph, int class_field_flag,
       void (*function) (IR_node_t node))'
```

is generated when option '-traverse' is present in the SPRUT command line. This function traverses graph given by its node 'graph' depth first (it means bottom up for tree). Arcs represented by class

fields are included in this graph if the second parameter value is nonzero. At every node a function given as parameter is executed. Arcs implementing lists of double linked fields are not traversed. The function does nothing if the 'graph' parameter value is equal to NULL.

20. Function

```
'void IR_traverse_reverse_depth_first
      (IR_node_t graph, int class_field_flag,
       int (*function) (IR_node_t node))'
```

is generated when option '-reverse-traverse' is present in the SPRUT command line. This function is analogous to the previous but the graph is traversed in reverse depth first order (it means top down order for tree). The traverse of children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All graph is traversed when the returned value is always nonzero.

21. Function

```
'IR_node_t IR_transform_dag
   (IR_node_t graph, int class_field_flag,
    int (*guard_function) (IR_node_t node),
    IR_node_t (*transformation_function) (IR_node_t node))'
```

is generated when option '-transform' is present in the SPRUT command line. This function is used for transformation of directed acyclic graphs. This is needed for various goals, e.g. for constant folding optimization. At first the function calls guard function. If the guard function returns zero, the function return the first parameter. Otherwise when the guard function returns nonzero, all fields (including class fields if parameter 'class_field_flag' is nonzero) of given node are also processed, i.e. the fields values are changed. And finally the node itself is processed by the transformation function (remember that the transformation function is never called for nodes which are parts of a graph cycle), and the result of transformation function is the result of given function.

22. Function

```
'void IR_start (void)'
```

is generated in any case. This function is to be called before any work with internal representation. The function initiates internal representation storage management (see next section), fulfills class field type specific initializations, evaluates actions bound to class fields, and make some other initiations.

23. Function

```
'void IR_stop (void)'
```

is generated in any case. The call of this function is to be last. The function fulfills class field type specific finalizations and finishes internal representation storage management.

3.2 C++ SPI

SPRUT generates C++ SPI only when option '-c++' is present on the command line. Mainly C++ SPI is analogous to one on C. Major difference is that 'IR_node_t' and 'IR_double_link_t' are pointer to classes not to structures. As the consequence, the most of functions are public or friend functions of the classes.

3.2.1 C++ SPI types, variables, macros

C++ interface file of SPI consists of C++ declarations of the following types, variables.

1. Type 'IR_node_mode_t' is represented by enumeration definition. There is enumeration constant with node type name and prefix 'IR_NM_' for each node type declaration even for abstract node type. Also there are enumeration constants with names 'IR_NM__root' and 'IR_NM__error' for and predefined type '%root' and for error nodes. 'IR_NM' is abbreviation of internal representation node mode. There is also the enumeration name 'IR_node_mode_enum' which is convenient to use in imported section for forward definition of pointer to this enumeration.

2. Type 'IR_node_t' represents a node (more correctly pointer to a class representing node). There are the type value 'NULL' reserved to represent none node. The class of this type is opaque (i.e. there are not public variables in the class) and all work with nodes are to be fulfilled through public or friend functions. But to show efficacy of types implementation it should be said that every node type is turned into a structure declaration. All these structures have first member of type 'IR_node_mode_t'. Other members represent the node fields except for SPRUT class field. Depending on SPRUT command line option '-flat-structure' (see SPRUT usage) this structure can contain also members corresponding to fields declared in super types of given node type or member which type is structure corresponding to immediate super type of given node type. SPRUT class fields for each non-abstract node type are represented by only one separate structure.

   The fields in different node types with the same name (when it is consequence of existing common super type with this field) have the same displacement from the structure begin because types of previous fields is guaranteed to be the same in these structures. The fields in different node types with the same name (when it is not consequence of existing common super type with this field) may have the different displacement from the structure begin therefore access to such fields requires additional indirect addressing.

   Double linked field (see internal representation description language) requires more memory (about in five times) for its implementation. Speed of SPI work with double linked fields is sightly slower. Because additional work is needed only to setting up the field value and my experience shows that usually number of setting up fields value is less than number of their fetching about in ten times.

   Type 'IR_node_t' is simply pointer to class representing node type '%root'. The name of this class is 'IR_node'. This name can be used in imported section for forward reference for nodes (instead of 'IR_node_t').

3. Optional variable 'IR_node_name' is array mapping values of type 'IR_node_mode_t' to names (strings) in C++ (see also option -no-node-name in SPRUT usage).

4. Variable 'IR_node_size' is array mapping values of type 'IR_node_mode_t' to size of corresponding structures on C++.

3.2.2 C++ Functions

C++ SPI interface file also contains definitions of generated functions to quering characteritics of nodes, creation, deletion, modification, comparison, copy, check, print, and binary input/output of internal representation nodes and functions for access to values of internal representation node fields.

These functions do not permit to work with abstract nodes. Only one graph function can be active, e.g. call of function 'IR_check_graph' is not permitted during work of any traverse or transformation function.

For sake of efficacy, the part of functions (access to internal representation node fields and quering charateristics of the nodes) is generated as inline.

1. Function 'IR_is_type (IR_node_mode_t type, IR_node_mode_t super)' which is public static (i.e. which is relative to class not to a object of the class) function of class 'IR_node' is evaluated as nonzero if the first argument of type 'IR_node_mode_t' is equal to the second argument or node type represented by the first argument is a sub-type of node type represented by the second argument. Otherwise the function returns zero. But this function returns always zero for any error node, i.e. this function can be used to define that the node of the first mode has fields of the node of the second mode.

2. Function 'IR_node_mode (void)' declared as public in the class 'IR_node' returns node mode represented by value of type 'IR_node_mode_t' for given node.

3. Function 'IR_is_of_type (IR_node_mode_t super)' declared as public in the class 'IR_node' is abbreviation of frequently used 'IR_is_type (node->IR_node_mode (), super)'.

4. Function 'IR_node_level (void)' declared as public in the class 'IR_node' has value which is internal representation description level in which declaration of given node type starts.

5. Access functions. These functions declared as public in the class 'IR_node' are generated when option '-access' is present in the SPRUT command line. There is one function for each field with unique name. Access function has name formed from prefix 'IR_' and corresponding field name. Access function has no arguments. Type of returned value is predefined type (if the field is of such type) or 'IR_node_t' (if the field is of a node type). Function declared as public in the class 'IR_node'

   ```
   'IR_double_link_t IR__first_double_link (void)'
   ```

   and functions declared as public in the class for which pointer of type 'IR_double_link_t' refers

   ```
   'IR_double_link_t IR__next_double_link (void)',

   'IR_double_link_t IR__previous_double_link (void)'
   ```

   return pointer to fields which have double links to given node. The node itself which contains such links can be determined by function declared as public in the class for which pointer of type 'IR_double_link_t' refers

   ```
   'IR_node_t IR__owner (void)'.
   ```

   Here type 'IR_double_link_t' represents pointer to class representing double fields. The functions may also check up that the node has given field (see SPRUT options).

6. Modification functions. These functions declared as public in the class 'IR_node' are generated when option '-set' is present in the SPRUT command line. There is one functions for each field with unique name. Modification function has name formed from prefix 'IR_set_' and corresponding field name. Modification function has one argument is of predefined type (if the field is of such type) or of type 'IR_node_t' (if the field is of a node type). Argument represents new value of the field.

   The function may check up constraint if the field represents arc to node of a node type or its sub-types (see SPRUT options). The functions may also check up that the node has given field.

There is also function declared as public in the class for which pointer of type 'IR_double_link_t' refers for modification of double links

```
'void IR__set_double_link (IR_node_t value)'
```

Remember that values of 'IR__next_double_link' and 'IR__previous_double_link' are changed after calling this function.

7. Function declared as friend of class 'IR_node'

   ```
   'IR_node_t IR_create_node (IR_node_mode_t node_mode)'
   ```

   is generated in any case. This function allocates storage for node of any type (except for abstract) represented by the argument (it can be also 'IRNM__error), sets up node mode and node level, fulfills fields assignments (except for class fields) in the same order as the fields are declared. The field assignment is fulfilled according to type specific operation (see section 'Type specific macros') and actions which are defined after given field in sections of skeleton and other fields of corresponding node type. After that the function returns the created node. The single way to create error nodes is to use this function with argument 'IRNM__error'. Remember that class 'IR_node' has no public constructors and destructors.

8. Node type specific creation functions declared as friend of class 'IR_node'. These functions are generated when option '-new' is present in the SPRUT command line. There is one function for each node type (except for abstract). Names of functions are formed from prefix 'IR_new_' and corresponding node type name. The number of parameters of a function is equal to one of skeleton fields of corresponding node type. The parameters of these functions have to be given in the order of the skeleton fields in the specification. Fields of the super-type (recursively) precede the ones of the sub-type. These functions may check up constraints on skeleton fields which represent arc to node of a node type or its sub-types (see SPRUT options). These functions call node creation function and after that assign parameters values to corresponding skeleton fields. After that the functions return the created node. These functions provide a mechanism similar to record aggregates. Nested calls of such functions allow programming with terms as in LISP.

9. Function declared as friend of class 'IR_node'

   ```
   'void IR_free_node (IR_node_t node)'
   ```

   is generated when option '-free' or '-free-graph' is present in the SPRUT command line. This function deletes given node. The deletion consists of evaluation of finalization macros for node fields (see field type specific macros) in reverse order and all node space deallocation. The function also delete double linked fields of given node from the corresponding lists of double linked fields. But the user should know that there may be dangling references to given node itself after the function call. Remember also that class 'IR_node' has no public constructors and destructors.

10. Function declared as public in the class 'IR_node'

    ```
    'void IR_free_graph (void)'
    ```

    is generated when option '-free-graph' is present in the SPRUT command line. This function deletes given node and all nodes accessible from given node. The nodes are deleted in depth first order.

11. Function declared as public in the class 'IR_node'

```
'void IR_conditional_free_graph
        (int (*guard) (IR_node_t node))'
```

is generated when option '-free-graph' is present in the SPRUT command line. This function deletes given node and all nodes accessible from given node. The nodes are deleted in depth first order. The processing (traversing and deleting) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All graph is deleted when the returned value is always nonzero.

12. Function declared as public in the class 'IR_node'

```
'IR_node_t IR_copy_node (void)'
```

is generated when option '-copy' or '-copy-graph' is present in the SPRUT command line. This function makes copy of given node. Field type specific macros are used (see section 'Type specific macros') to make this.

13. Function declared as public in the class 'IR_node'

```
'IR_node_t IR_copy_graph (void)'
```

is generated when option '-copy-graph' is present in the command line. This function makes copy of graph given by its node.

14. Function declared as public in the class 'IR_node'

```
'IR_node_t IR_conditional_copy_graph
                    (int (*guard) (IR_node_t node))'
```

is generated when option '-copy-graph' is present in the command line. This function makes copy of graph given by its node. The processing (traversing and copying) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All nodes of graph are copied when the returned value is always nonzero.

15. Function declared as public in the class 'IR_node'

```
'int IR_is_equal_node (IR_node_t node_2)'.
```

is generated when option '-equal' or '-equal-graph' is present in the SPRUT command line. This function returns nonzero iff given node is equal to 'node_2', i.e. given node and node_2 have the same type and all fields of given node are equal to the corresponding fields of 'node_2'. Field type specific macros are used (see section 'Type specific macros') to make this.

16. Function declared as public in the class 'IR_node'

```
'int IR_is_equal_graph (IR_node_t graph_2)'
```

is generated when option '-equal-graph' is present in the SPRUT command line. This function returns nonzero iff graph starting with given node is equal to graph starting with node 'graph_2', i.e. the graphs have the same structure and all predefined type fields of given node are equal to the corresponding fields of 'graph_2'.

17. Function declared as public in the class 'IR_node'

```
'int IR_is_conditional_equal_graph
       (IR_node_t graph_2, int (*guard) (IR_node_t node))'
```

is generated when option '-equal-graph' is present in the SPRUT command line. This function returns nonzero iff graph starting with given node is equal to graph starting with node 'graph_2', i.e. the graphs have the same structure and all predefined type fields of given node are equal to the corresponding fields of 'graph_2'. The processing (traversing and comparing) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All nodes of the graphs are compared when the returned value is always nonzero. Unprocessed graph nodes are believed to be equal.

18. Function declared as public in the class 'IR_node'

```
'int IR_check_node (int class_field_flag)'
```

This function is generated when option '-check' or '-check-graph' is present in the SPRUT command line. By default the most of generated functions which work with internal representation nodes do not check up constraints which described in the specification. There are two kinds of constraints. The first is constraint on field which represents arc to node of given type or its sub-types. The second is constraint which defined by target code (see the internal representation definition language). The user is responsible to adhere to the constraints. Function 'IR_check_node' can be used to check all constraints corresponding to given node. In case of a constraint violation the involved node and all child nodes are printed on standard error and the function returns nonzero (error flag). If the parameter value is zero then class fields of given node are not checked up.

19. Function declared as public in the class 'IR_node'

```
'int IR_check_graph (int class_field_flag)'
```

is generated when option '-check-graph' is present in the SPRUT command line. This function checks given node and all nodes accessible from given node. It should be known that a class field is processed only when the parameter value is nonzero. In case of fixing any constraint violation returns nonzero (error flag).

20. Function declared as public in the class 'IR_node'

```
'int IR_conditional_check_graph
       (int class_field_flag,
        int (*guard) (IR_node_t node))'
```

is generated when option '-check-graph' is present in the SPRUT command line. This function checks given node and all nodes accessible from given node. It should be known that a class field is processed only when the first parameter value is nonzero. In case of fixing any constraint violation returns nonzero (error flag). The processing (traversing and checking) children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All nodes of the graph are checked when the returned value is always nonzero. It is better to free changed nodes which are not used after transformation especially when there are double links to the nodes (otherwise the nodes can be accessible with the aid of functions for work with double links).

21. Function declared as public in the class 'IR_node'

```
'void IR_print_node (int class_field_flag)'
```

is generated when option '-print' is present in the SPRUT command line or options '-check' or 'check-graph' is present because check functions use this print function. The function outputs values of node fields with their names to standard output in readable form. To make this the function outputs

sequentially all the node fields (including class fields if the parameter value is nonzero) with the aid of field type specific macros (see below).

22. Function declared as public in the class 'IR_node'

```
'int IR_output_node (FILE *output_file, int level)',
```

and function declared as friend of class 'IR_node'

```
'IR_node_t IR_input_node (FILE *input_file,
                          IR_node_t *original_address)'
```

are generated when option correspondingly '-output' or '-input' is present in the SPRUT command line. Function 'IR_output_node' writes node to given file. To make this the function outputs given node address and after that sequentially all the node fields (except for class fields) whose declarations are present in internal representation description of 'level' or less than the one (see internal representation description language). The node type must be also declared in internal representation description of 'level' or less than the one. A nonzero function result code means fixing an error during output.

Function 'IR_input_node' reads node from given file, allocates space for the node, initiates its fields and changes the node fields output early. To make this the function uses function 'create_node'. The output level of input node must be less or equal to the internal representation level of given SPI. Null value returned by the function means fixing an error during input. The function returns original address of input node through the second parameter. The output and input of fields is fulfilled with the aid of field type specific macros (see section 'Type specific macros').

23. Function declared as public in the class 'IR_node'

```
'void IR_traverse_depth_first
      (int class_field_flag,
       void (*function) (IR_node_t node))'
```

is generated when option '-traverse' is present in the SPRUT command line. This function traverses graph given by its node in depth first order (it means bottom up for tree). Arcs represented by class fields are included in this graph if the first parameter value is nonzero. At every node a function given as parameter is executed. Arcs implementing lists of double linked fields are not traversed.

24. Function declared as public in the class 'IR_node'

```
'void IR_traverse_reverse_depth_first
      (int class_field_flag,
       int (*function) (IR_node_t node))'
```

is generated when option '-reverse-traverse' is present in the SPRUT command line. This function is analogous to the previous but the graph is traversed in reverse depth first order (it means top down order for tree). The traverse of children of node which is passed to the parameter-function is not fulfilled if the parameter-function returns zero. All the graph is traversed when the returned value is always nonzero.

25. Function declared as public in the class 'IR_node'

```
'IR_node_t IR_transform_dag
   (int class_field_flag,
    int (*guard_function) (IR_node_t node),
    IR_node_t (*transformation_function) (IR_node_t node))'
```

is generated when option '-transform' is present in the SPRUT command line. This function is used for transformation of directed acyclic graphs. This is needed for various goals, e.g. for constant folding optimization. At first the function calls guard function. If the guard function returns zero, the function return given node. Otherwise when the guard function returns nonzero, all fields (including class fields if parameter 'class_field_flag' is nonzero) of given node are also processed, i.e. the fields values are changed. And finally the node itself is processed by the transformation function (remember that the transformation function is never called for nodes which are parts of a graph cycle), and the result of transformation function is the result of given function.

26. Function

    `'void IR_start (void)'`

    is generated in any case. This function is to be called before any work with internal representation. The function initiates internal representation storage management (see next section), fulfills class field type specific initializations, evaluates actions bound to class fields, and make some other initiations.

27. Function

    `'void IR_stop (void)'`

    is generated in any case. The call of this function is to be last. The function fulfills class field type specific finalizations and finishes internal representation storage management.

3.3 Type specific macros

C and C++ SPI uses some macros which work with node fields. These macros are field type specific and are generated by SPRUT for each predefined types and type 'IR_node_t'. All these macros can be redefined. There are the following field type specific macros.

1. Initialization and finalization macros. These macros can assign an initial value to field or/and even construct complex data structures.

   Finalization is performed immediately before the node is deleted. For example finalization macro can release memory space dynamically allocated by initialization macro. These macros have names formed from prefixes correspondingly 'IR_BEGIN_' and 'IR_END_' and name of the field type, e.g.

   'IR_BEGIN_integer' or 'IR_END_IR_node_t'.

   By default initialization macros have definition

   `'bzero ( sizeof (a))'`

   where 'a' is the macros argument. Finalization macros do nothing. It is very important for correct work of double linked fields that initialization macro for 'IR_node_t' initiates a pointer by NULL value.

2. Copy macros. The operation copy is used by copy functions generated by SPRUT. These macros have names formed from prefix 'IR_COPY_' and name of the field type. The default macros definitions are

   `'((a) = (b))'`

where 'a' and 'b' are the macro parameters representing fields of given type. Therefore, pointer semantics is assumed for fields of a pointer type. The user has to take care about the macros redefinition if value semantics is needed.

3. Equality macros. These macros check whether two fields are equal. They are used in functions which test the equality of nodes. These macros have names formed from prefix 'IR_EQUAL_' and name of the type. The default macros definitions are

   ```
   '((a) == (b))'
   ```

   where 'a' and 'b' are the macro parameters representing fields of given type.

4. Print macros. These macros are used by print function generated by SPRUT. These macros have names formed from prefix 'IR_PRINT_' and name of the type. By default the macros for 'IR_node_t' is defined as

   ```
   'printf ("%lx", (unsigned long) (a))'
   ```

   where 'a' is macro parameter. The macros for other types do nothing.

5. Input/output macros. These macros are used by input/output functions generated by SPRUT. These macros have names formed from prefixes 'IR_INPUT_' and 'IR_OUTPUT_' and name of the type. By default the macros have definitions

   ```
   '(fwrite ( sizeof (field), 1, (file))
                    != sizeof (field))'

   and '(fread ( sizeof (field), 1, (file))
                    != sizeof (field))'
   ```

   where 'file' and 'field' are the macros parameters. The definitions may be more complicated for predefined types.

It should be repeated that it is possible to redefine the operations by including new macro definitions in the C/C++ declaration section. For example

```
%import{
typedef int PT_line, PT_position;
}
%local{
#define IR_BEGIN_PT_line(l) (l) = ...
#define IR_BEGIN_PT_position(p) (p) = ...
#define IR_BEGIN_IR_node_t (n) (n) = ...
}
%type PT_line PT_position PT_boolean

%%
%abstract
node :: %root
  line : PT_line
  position : PT_position
...
```

3.4 Storage management macros

SPRUT completely automatically generates macros of storage management for the nodes. Usually, the user does not have to care about it. The predefined storage management uses C/C++ standard functions for global storage with free lists. The default macro definitions are placed before default field type specific macro definitions but after all code insertions. The user can create own storage manager by redefinition of one or more the following macros and placing them in a C/C++ declaration section. But the user should know that all functions generated by SPRUT believe that the storage can not change its place after the allocation.

1. Macros 'IR_START_ALLOC()', 'IR_STOP_ALLOC()' are evaluated in functions correspondingly 'IR_start' and 'IR_stop' and are to be used to start and finish work with the storage manager.
2. Macros 'IR_ALLOC(ptr, size, ptr_type)', 'IR_FREE(ptr, size)' allocate and free storage whose size is given as the second parameter. The first parameter serves to return pointer to memory being allocated or to point to memory being freed. The third parameter is type of 'ptr'.

By default these macros are defined as follows:

```
#define IR_START_ALLOC()
#define IR_STOP_ALLOC()
#define IR_ALLOC(ptr, size, ptr_type)\
                            ((ptr) = (ptr_type) malloc (size))
#define IR_FREE(ptr, size)   free (ptr)
```

4 SPRUT Usage

```
SPRUT(1)                        User Manuals                        SPRUT(1)



NAME
       sprut - internal representation description translator

SYNOPSIS
       sprut  [  -c++  -v  -macro  -only-macro  -debug  -pprefix -no-line -all
       -access -set -new -free -free-graph -copy  -copy-graph  -equal  -equal-
       graph  -check  -check-graph -print -input -output -traverse -transform]
       specification-file


DESCRIPTION
       SPRUT (internal representation definition translator)  generates  stanâĂŘ
       dard procedural interface ( SPI ) for work with internal representation
       which is described in specification file.  The specification file  must
       have suffix '.sprut'.
         If  the specification file is an extension (see language description)
       of another specification file the later is also used to SPI  generation
       and so on.
```

SPI consists of interface and implementation files having the same names as one of specification file and correspondingly suffixes ‘.h’ and ‘.c’ (C code) or ‘.cpp’ (C++ code).

Full documentation of SPRUT and SPI see in SPRUT User’s manual.

OPTIONS

The options which are known for SPRUT are:

-c++ Output of C++ code instead of C code (which is default).

-v SPRUT outputs verbose warning information (about fields with the same name in different node types, about repeated declaration of a predefined type and so on) to standard error stream.

-statistics
SPRUT outputs statistic information (about number of (all, abstract, double) node types, number of (all, double, synonym, class, skeleton, others) fields) to standard output stream.

-flat SPRUT generates code for flat work with node fields. It means that node fields declared in an abstract type are processed in each place where fields of corresponding sub-type are processed. By default SPRUT generates code which processes node fields recursively (i.e. code for processing (e.g. copying) fields declared in super types exists in one exemplar. The option usage generates more fast code, but the code is considerably bigger.

-flat-structure
SPRUT generates flat implementation of C/C++ structures correâĂŘ sponding to node types, i.e. node type is represented by C/C++ structure which contains members corresponding to all node fields including fields declared in super types of given node type. By default SPRUT generates the structures which contain only members corresponding to fields declared only in given node type and member whose type is structure corresponding to immediâĂŘ ate super type of given node type. This considerably decreases number C/C++ declarations in interface file especially with many node types with multi-level inheritance.

-fast This option forces to generate some tables as unpacked and as a consequence to speed up code for access to fields. By default the tables are generated as packed. This results in considerâĂŘ able memory economy.

-long-node-size
IRTD generates many long tables which contain displacements relâĂŘ ative to begin of C/C++ structures which implement node types. This option usage forces to represent the displacements as unsigned long. As a consequence any size nodes can be used. By

default it is believed that size of structure implementing a node is represented by byte, i.e. maximal size of C/C++ strucâĂŘ tures which implement nodes is supposed to be not greater than 255 bytes and as a consequence much memory is economized. It should be remember that function ‘IR_start’ (see SPI) generated in debugging mode checks correctness of real node sizes.

-short-node-size
This option usage forces to represent the displacements in structures implementing node types as unsigned short. See also option ‘-long-node-size’.

-macro SPRUT generates access macros and functions (see SPI). Only option ‘-macro’ or ‘only-macro’ can be in SPRUT command line.

-only-macro
SPRUT generates macros instead of access functions (see SPI).

-debug SPRUT generates checking constraint determined by type sub-type relations in node field declaration in modification functions (see SPI). Also other checks (e.g. that a node has given field and others) are fulfilled by SPI functions (but not macros) genâĂŘ erated under this option.

-pprefix
Generated SPI uses ‘prefix’ instead of ‘IR_’ for names of SPI objects.

-no-line
SPRUT does not generate files containing numbered line direcâĂŘ tives.

-all SPRUT generates all SPI functions. Its effect is equivalent to presence of all subsequent options in the command line.

-access
SPRUT generates access functions (see SPI).

-set SPRUT generates modification functions (see SPI).

-new SPRUT generates node type specific creation functions (see SPI).

-free SPRUT generates function for deletion of nodes (see SPI).

-free-graph
SPRUT generates functions for deletion of graphs and nodes (see SPI).

-copy SPRUT generates function for copying nodes (see SPI).

-copy-graph

```
       SPRUT generates functions for copying graphs and nodes (see
       SPI).

-equal SPRUT generates function for determination of nodes equality
       (see SPI).

-equal-graph
       SPRUT generates functions for determination of nodes and graphs
       equality (see SPI).

-check SPRUT generates function for checking node constraints (see
       SPI).

-check-graph
       SPRUT generates functions for checking nodes and graphs conâĂŘ
       straints (see SPI).

-print SPRUT generates function for printing nodes (see SPI).

-input SPRUT generates function for input of nodes (see SPI).

-output
       SPRUT generates function for output of nodes (see SPI).

-traverse
       SPRUT generates function for traversing graphs (see SPI).

-reverse-traverse
       SPRUT generates function for reverse traversing graphs (see
       SPI).

-transform
       SPRUT generates functions for transforming acyclic graphs (see
       SPI).

-no-node-name
       SPRUT does not generate array containing node names (see SPI) if
       it is not necessary. For example, the array is necessary when
       function for print of nodes are generated.

FILES
       file.sprut
              SPRUT specification file
       file.c
              generated SPI implementation file
       file.cpp
              generated C++ implementation file
       file.h
              generated SPI interface file

       There are no any temporary files used by SPRUT.
```

```
ENVIRONMENT
       There are no environment variables which affect SPRUT behavior.

DIAGNOSTICS
       SPRUT diagnostics is self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@gcc.gnu.org

SEE ALSO
       msta(1), shilka(1), oka(1), nona(1).  SPRUT manual.

BUGS
       Please, report bugs to https://github.com/dino-lang/dino/issues.



COCOM                              5 APR 2001                           SPRUT(1)
```

5 Appendix 1 - Syntax of internal representation description language

YACC notation is used to describe full syntax of internal representation description language.

```
 %token PERCENTS COMMA COLON DOUBLE_COLON SEMICOLON
       IDENTIFIER CODE_INSERTION EXPRESSION ADDITIONAL_C_CODE

%token DOUBLE EXTEND LOCAL IMPORT EXPORT TYPE ROOT ABSTRACT CLASS
       SKELETON OTHER

%start description

%%

description : declaration_part PERCENTS node_type_definition_list
              ADDITIONAL_C_CODE
            ;

declaration_part :
                 | declaration_part DOUBLE any_node_type_name
                 | declaration_part EXTEND IDENTIFIER
                 | declaration_part predefined_types_declaration
                 | declaration_part LOCAL CODE_INSERTION
                 | declaration_part IMPORT  CODE_INSERTION
                 | declaration_part EXPORT  CODE_INSERTION
                 ;

predefined_types_declaration : TYPE
```

```
                            | predefined_types_declaration IDENTIFIER
                            ;

node_type_definition_list : node_type_definition
                          | node_type_definition_list
                              SEMICOLON  node_type_definition
                          ;

node_type_definition :
                     | abstract_node_flag type_nodes_identifier_list
                          optional_immediate_super_type_list
                          class_field_definition_part
                          skeleton_field_definition_part
                          other_field_definition_part
                     ;

abstract_node_flag :
                   | ABSTRACT
                   ;

type_nodes_identifier_list : identifier_list
                           ;

optional_immediate_super_type_list :
                                   | immediate_super_type_list
                                   ;

immediate_super_type_list : DOUBLE_COLON any_node_type_name
                          | DOUBLE_COLON COMMA any_node_type_name
                          | immediate_super_type_list COMMA
                             any_node_type_name
                          ;

any_node_type : ROOT
              | IDENTIFIER
              ;

class_field_definition_part :
                            | CLASS field_definition_list
                            ;

skeleton_field_definition_part :
                               | SKELETON field_definition_list
                               ;

other_field_definition_part :
                            | OTHER field_definition_list
                            ;

field_definition_list :
                      | field_definition_list field_definition
```

```
                    | field_definition_list constraint
                    | field_definition_list action
                    ;

field_definition : field_identifier_list  COLON  optional_double
                     any_node_type_name
                 ;

optional_double :
                | DOUBLE
                ;

constraint : EXPRESSION
           ;

action : CODE_INSERTION
       ;

field_identifier_list : identifier_list
                      ;

identifier_list : IDENTIFIER
                | identifier_list  COMMA  IDENTIFIER
                ;
```