Parsing The Ada Programming Language

February 22, 2007

Marc Mooser, mooser@students.unibe.ch

Technical Report for a Bachelor Project at the

University of Berne, Switzerland

Software Composition Group

Abstract

In this technical report we show how to produce a parser for the Ada 83 programming language [2]. It features general ideas about parsing and the parser definitions for main parts of Ada 83. Using SmaCC (Smalltalk Compiler Compiler, [4]), a LR-parser-generator, we produce different parsers which are able to parse different parts or also the full language. Our SmaCC-version runs in Visual Works Smalltalk [3] and we show how our parsers are compiled there, how one can use Smalltalk programming to produce a syntax tree out of the source code and also fetch and process the structure of these codes. The structure and corresponding contents is imported as models into Moose [5] — a tool to measure, analyze, visualize and reengineer large amounts of source codes (applications) written in different programming languages in a abstracted way using their concrete model. And there is a discussion about problems one encounters when trying to find exact machine-directives to parse texts in general, how one may fix them and what specific problems arised while this project.

Contents

1 Introduction **3**

2 Short example of Ada code **4**

3 Grammar of Ada 83 **5**
3.1 Definitions top-down, first part 5
3.2 Simple Statements 8
3.3 Compound Statements 9
3.4 Object Declarations 9
3.5 Type Declarations 10
3.6 Subtype Declarations 10
3.7 Expressions 11
3.8 Names 11
3.9 The Scanner 12

4 ParseTree **13**

5 Exporting to Moose and Mondrian **13**
5.1 Exporting to Mondrian 14

6 Problems, Troubleshooting and Testing **15**
6.1 Shift/Reduce conflicts 15
6.2 Reduce/Reduce conflicts 16
6.3 Other Problems and Troubleshooting 16
6.4 Testing 16

A Appendix section **16**
A.1 About the different coverages of parsers — from arbitary texts to texts with special structure 16

B Small examples **17**
B.1 An Example: Expressions 18
B.2 SyntaxTree of the example 19

1 Introduction

In order to be able to analyse source code one has first to *parse* it which means to extract the specific structural elements out of it. Therefore one needs a grammar of the corresponding programming language which will help the *parser* to translate a source code into a logical model.

Of course the language should be already defined in some way. But a parser needs a set of definitions of all the different elements which can occur in the source and their relationships — their possible content and their nesting. Typically in form of a syntax tree which outlines implicitly all the different possible concrete source codes of a specific language. This syntax tree may then serve at the same time as a formal definition of this programming language.

Typically one uses *parser generators* which produce for every definition another specific parser, for example as a native computer program. That will then be called a *parser* for a specific programming language (or other sets of predefined texts). The same mechanism can be used to produce a compiler for a programming language since such a compiler needs first to parse the source text, too. Such a extended parser generator which generates a compiler would therefore be called a *compiler compiler*.

In the following we present possible problems, an Ada 83 example code, the grammar of Ada 83 and the parse tree and how we can import the code into Moose and Mondrian. In the appendix we give a short overview about parsing in general and a small example together with its parse tree.

2 Short example of Ada code

```
- - A comment begins with two dashes
- - Context Clauses:
with Except;
with Screen_Output;
with Stack;
with Values;

procedure Hello is
  - - Definitions before begin:

    - - Definitions of variables and types
    I : Integer;
    MAX : constant := 5;
    type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
    subtype RED_BLUE is RAINBOW;
    subtype SMALL_INT is INTEGER range 5 .. 10;

    - - Definition of a procedure
    procedure P1 is
    begin
      Put;
    end P1;

    - - Definition of a function
    function F1(X : in out INTEGER) return BOOLEAN is
    begin
      Put;
      return X * X;
    end F1;

begin
  - - Body of the procedure hello
  null; – null statement
  var1 := var2 and 4;  - - assignment statement
  Putline;           - - procedure call statement
  if null = null then   - - if statement
    null;
  end if;
  if 4 = 5 then   - - if statement
  Putline;
  end if;
  if (4 = 5) and (6>7) then   - - if statement
    Putline;
  end if;
  while (15>4) or (4<3) or (5=5) loop   - - while statement
    Putline;
  end loop;
  case aCase is      - - case statement
    when A => Putline;
    when B => Putline;
  end case;
  loop
    Putline;
  end loop;
end Hello;
```

This example starts with Context Clauses which have similar meaning as include commands in C. The main part consist of a subprogram which can be a function or a procedure. Here it is a procedure with the name hello. The actual code of it is listed between the last begin and end. But between the head of the procedure and its body

there is an optional section which is called declarative part. It may contain different kind of declarations, but also definitions of procedures or functions.

3 Grammar of Ada 83

3.1 Definitions top-down, first part

SmaCC always tries to resolve the first definition, so we have to start with program.

```
goal:                       program 'program' { Ada.Program withElements: program. };
program:                    (contextclause | compilation_unit)*;
compilation_unit:           subprogram_body |
                            subprogram_declaration |
                            package_declaration |
                            package_body |
                            generic_declaration |
                            generic_instantiation |
                            subunit;

contextclause:              context_clause 'contextclause' { Ada.ContextClause withElements: contextclause. };
context_clause:             with_clause use_clause*;
with_clause:                "with" <identifier> ("," <identifier>)*";";
use_clause:                 "use" <identifier>";";

subprogram_body:            subprogram_procedure |
                            subprogram_function;
subprogram_procedure:       "procedure" <identifier> 'id' [formal_part] "is"
                            declarative_part 'decl_part'
                            body 'body'
                            <identifier> ";"
                            {Ada.Procedure withTheDeclarativePart: decl_part withElements: body withName: id value.};
subprogram_function:        "function" designator 'id' [formal_part] "return" type_mark "is"
                            declarative_part 'decl_part'
                            body 'body'
                            <identifier> ";"
                            {Ada.Function withTheDeclarativePart: decl_part withElements: body withName: id value.};

subprogram_declaration:     subprogram_specification;
subprogram_specification    "procedure" identifier [formal_part] |
                            "function" designator [formal_part] "return" type_mark;
designator:                 <identifier> 'id' {id value} | operator_symbol 'id' {id value};
operator_symbol:            <string_lit>;
```

```
package_body:            "package" "body" <pkg_simple_name> "is"
                         declarative_part
                         body
                         <pkg_simple_name> ";"
                         {Ada.Package withTheDeclarativePart: decl_part withElements: body withName: id value.};
package_declaration:     "package" <pkg_id> "is"
                         (basic_declarative_item)*
                         ("private" (basic_declarative_item)* )?
                         end <identifier> ";" ;
generic_declaration:     generic_1 | generic_2;
generic_1:               generic_formal_part subprogram_specification ;
generic_2:               generic_formal_part package_specification;
subunit:                 "seperate" "(" parent_unit_name ")" proper_body ;
```

The definitions before begin

```
declarative_part:               (basic_declarative_item)* (later_declarative_item)*;
basic_declarative_item:         basic_declaration | representation_clause | use_clause | pragma;
basic_declaration:              object_declaration | number_declaration |
                                type_declaration | subtype_declaration |
                                subprogram_declaration | package_declaration |
                                task_declaration | generic_declaration |
                                exception_declaration | generic_instantiation|
                                renaming_declaration | deferred_constant_declaration;
later_declarative_item:         body2 |
                                subprogram_declaration | package_declaration|
                                task_declaration | generic_declaration|
                                use_clause | generic_instantiation ;
body2:                          proper_body | body_stub;
body_stub:                      subprogram_specification "is" "separate" ";" |
                                "package" "body" package_simple_name "is" "separate" ";" |
                                "task" "body" task_simple_name "is" "separate" ";" ;
proper_body:                    subprogram_body | package_body; #| task_body ;
```

Tasks

```
task_body:
task_body_2nd:                  [declarative_part]
                                "begin"
                                seq_of_statements
                                [exception
                                exception_handler
                                (exception_handler)*]
                                end [task_simple_name] ";" ;
task_declaration:               task_specification;
task_specification:             "task" [type] <identifier> ["is"
                                (entry_declaration)*
                                (representation_clause)*
                                "end" [task_simple_name]] ;
```

for subprograms, body

```
formal_part:                    "(" param_spec (";" param_spec)* ")";
param_spec:                     identifier_list ":" mode? type_mark (":=" expression)?;
identifier_list:                <identifier> ("," <identifier>)*;
mode:                           "in" | "out" | "in out";
body:                           begin_
                                seq_of_statements 'elem'
                                exception?
                                end_;
begin_:                         "begin" 'id' { Ada.BeginKeyword withElements: empty withName: id. };
end_:                           "end" 'id' { Ada.EndKeyword withElements: empty withName: id. };
```

3.2 Simple Statements

```
seq_of_statements:          statement*;
statement:                  comment | simple_statement | compound_statement;
simple_statement:           null_statement |
                            assignment_statement |
                            function_call |
                            exit_statement | return_statement |
                            goto_statement | entry_call_statement |
                            delay_statement | abort_statement |
                            raise_statement | code_statement;

exit_statement:             exit [<identifier>] ["when" expression]”;”; # loop_name, condition
assignment_statement:       assignment1 | assignment2;
assignment1:                identifier ’id’ ":=" expression ’expr’ ";"
                            { Ada.AssignStatement withId: id withElements: expr withName: id. };
assignment2:                inner_function_call ’id’ ":=" expression ’expr’ ";"
                            { Ada.AssignStatement withElements: expr withName: id. };
goto_statement:             "goto" <identifier>”;”;
entry_declaration:          "entry" <identifier> ["("discrete_range")"] [formal_part]”;”;
#entry_call_statement:      <identifier> [actual_parameter_part]”;”; #entry_name
accept_statement:           "accept" <identifier> ["("entry_index")"] [formal_part]
                            ["do"
                            seq_of_statements
                            "end" [<identifier>]]”;”;
entry_index:                expression”;”;
delay_statement:            "delay" simple_expression”;”;
abort_statement:            "abort" <identifier> ("," <identifier>)* ”;”;
raise_statement:            "raise" [<identifier>]”;”; #exception_name;
code_statement:             type_mark "’" <identifier>”;”; #record_aggregate;
null_statement:             "null" ’id’ ";" { Ada.NullStatement withElements: empty withName: ’noname’. };
return_statement:           "return" (expression)? ’expr’ ";"
                            { Ada.ReturnStatement withElements: empty withName: expr. };
```

function call

```
function_call:              <identifier> ’identifier’ [actual_parameter_part] ’function_arguments’ ";"
                            { Ada.FunctionCall withElements: function_arguments withName: identifier value. };
inner_function_call:        <identifier> ’identifier’ actual_parameter_part ’function_argument’
                            { Ada.FunctionCall withElements: function_argument withName: identifier value. };
actual_parameter_part:      "(" expression ("," expression)* ")";
```

3.3 Compound Statements

```
compound_statement:          if_statement |
                             for_statement |
                             loop_statement |
                             while_statement |
                             case_statement ;

if_statement:                "if" expression ’expr’ "then"
                             seq_of_statements ’seq’
                             "end" "if" ";"
                             { Ada.IfStatement withElements: seq withName: expr. };
case_statement:              "case" <identifier> ’id’ "is"
                             case_line* ’caselines’
                             "end" "case" ";"
                             { Ada.CaseStatement withElements: caselines withName: id. };
case_line:                   "when" <identifier> "=>" statement;
loop_statement:              "loop"
                             seq_of_statements ’seq’
                             "end" "loop" ";"
                             { Ada.LoopStatement withElements: ’seq’ withName: id. };
for_statement:               "for" loop_parameter_spec ’expr’ "loop"
                             seq_of_statements ’seq’
                             "end" "loop" ";"
                             { Ada.ForStatement withElements: ’seq’ withName: expr. };
while_statement:             "while" expression ’expr’ "loop"
                             seq_of_statements ’seq’
                             "end" "loop" ";"
                             { Ada.WhileStatement withElements: seq withName: expr. };
loop_parameter_spec:         <identifier> "in" ("reverse")? discrete_range ;
```

3.4 Object Declarations

```
object_declaration:          object_declaration_a | object_declaration_b;
object_declaration_a:        id_list ’id’ ":" ["constant"] [subtype_indication] ’elem’ [":=" expression] ";"
                             { Ada.ObjNrDeclaration withElements: elem withName: id };
object_declaration_b:        id_list ’id’ ":" ["constant"] constrained_array_def (":=" expression)? ";" ;
id_list:                     <identifier> ("," <identifier>)*;
```

3.5 Type Declarations

```
type_declaration:               full_type_declaration |
                                incomplete_type_declaration |
                                private_type_declaration ;
full_type_declaration:          "type" <identifier> 'id' (discriminant_part)?
                                "is" type_definition 'typedef' ";"
                                { Ada.TypeDeclaration withElements: typedef withName: id. } ;
type_definition:                enum_type_def | integer_type_def |
                                real_type_def | array_type_def |
                                record_type_def | access_type_def |
                                derived_type_def;
array_type_def:                 unconstrained_array_def | constrained_array_def;
unconstrained_array_def:        "array" "(" index_subtype_def (","
                                index_subtype_def)* ")" "of" component_subtype_indication;
index_subtype_def:              type_mark range "<>";
constrained_array_def:          "array" index_constraint2 "of" component_subtype_indication;
index_constraint2:              "(" discrete_range ("," discrete_range)* ")" ;
discrete_range:                 discrete_subtype_indication | range;
enum_type_def:                  "(" <identifier> ("," <identifier>)* ")" ;
incomplete_type_declaration:    "type" <identifier> [discriminant_part];
private_type_declaration:       "type" <identifier> [discriminant_part]
                                "is" ["limited"] "private";
```

3.6 Subtype Declarations

```
subtype_declaration:            "subtype" <identifier> 'is' "is" subtype_indication 'subtypeindi' ";"
                                { Ada.SubtypeDeclaration withElements: subtypeindi withName: id. } ;
subtype_indication:             type_mark (constraint)? ;
type_mark:                      type | subtype_name;
constraint:                     range_constraint | floating_point_constraint | fixed_point_constraint |
                                index_constraint | discriminant_constraint ;
range_constraint:               "range" range;
range:                          range_attribute | (simple_expression ".." simple_expression);
range_attribute:                <identifier> "'" "range";
index_constraint:               "(" simple_expression ".." simple_expression ")" ;

is_def:                         "is" list;
pragma:                         "pragma" <identifier> ["(" <identifier> ("," <identifier>)*")"] ";"
                                { Ada.Pragma withElements: empty withName: id. };
argument_assoc:                 [<identifier> "=>"] name #argument_identifier
                                | [<identifier> "=>"] expression; #argument_identifier
```

3.7 Expressions

```
expression:               relation
                          ( ("and" | "and then" | "or" | "or else" | "xor")
                          relation )*;
relation:                 simple_expression sndPart?;
sndPart:                  relational_operator simple_expression |
                          ("not")? ("in range" | "in" type_mark);
simple_expression:        (unary_adding_operator)?
                          term (binary_adding_operator term)*;
term:                     factor (multiplying_operator factor)*;
factor:                   primary ("**" primary)? | "abs" primary |
                          "not" primary | ":" primary | "=>" primary;
primary:                  aggregate | "null" | #because of eg. ’if null=null then’
                          <string_lit> | <numeric_lit> |
                          <identifier> |
                          <identifier>"’"<identifier>["("expression")"] |
                          expression"’"<identifier> | actual_parameter_part |
                          allocator | inner_function_call | inner_function_call"’"<identifier>
                          type_conversion |
                          qualified_expression | "("expression")" | "<quote>" expression "<quote>";

binary_adding_operator:   "+" | "-" | "&";
unary_adding_operator:    "+" | "-";
multiplying_operator:     "*" | "/" | "mod" | "rem";
relational_operator:      "=" | "/=" | "<" | "<=" | ">" | ">=";
```

3.8 Names

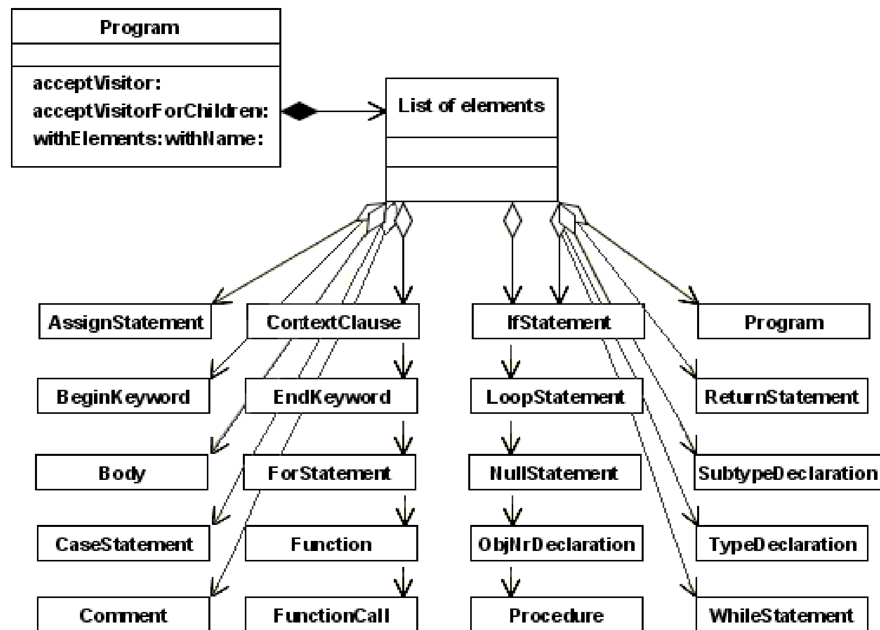
```
prefix:               name | function_call;
name:                 simple_name | <char_lit> |
                      operator_symbol | indexed_component |
                      slice | selected_component |
                      attribute;
simple_name:          <identifier>;
indexed_component:    prefix "(" expression ( "," expression )* ")" ;
slice:                prefix "(" discrete_range ")";
selected_component:   prefix "." selector;
selector:             simple_name | character_literal |
                      operator_symbol | "all";
attribute:            prefix "’" attribute_designator;
attribute_designer:   simple_name ("("universal_static_expression")")? ;
```

3.9 The Scanner

```
<eol>:                    \r | \n | \r\n;
<whitespace>:             \s+;
<letter>:                 [a-z]|[A-Z];
<digit>:                  [0-9];
<quote> :                 \";
<special_char>:           \# | & | ’ | \( | \) | \* | \+ | , | \- | \. | / | \: | \;| \< | = | \> | _ | \|;
<other_special_char>:     ! | $ | % | \? | @ | \[ | \\ | \] | ^ | ‘ | \{ | \}| ~;
<space>:                  \s;
<graphic_char>:           <letter> | <special_char> | <other_special_char> | <space>;
<identifier>:             <letter>((_|\.)?( <letter> | <digit> ))*;
<number> :                0 | [1-9][0-9]*;
<integer>:                <digit>((_)?<digit>)*;
<exponent>:               E(\+)?<integer> | E-<integer>;
<decimal_lit>:            <integer>(\.<integer>)?(<exponent>)?;
<extended_digit>:         <digit> | <letter>;
<based_integer>:          <extended_digit>((_)?<extended_digit>)*;
<numeric_lit>:            <decimal_lit>|<based_lit>;
<char_lit>:               ’<graphic_char>’;
<string_lit>:             "(<graphic_char>)*" | %(<graphic_char>)*%;
<pragma_arg>:             LIST | OPTIMIZE | INLINE | SUPPRESS;
```

4 ParseTree

The ParseTree gets customized by using instructions like { Ada.ContextClause withElements: contextclause. }; in the definitions. So program elements in a source text can get fetched and transformed into objects of the adequate program element class.



All of these program elements are subclasses of Program

5 Exporting to Moose and Mondrian

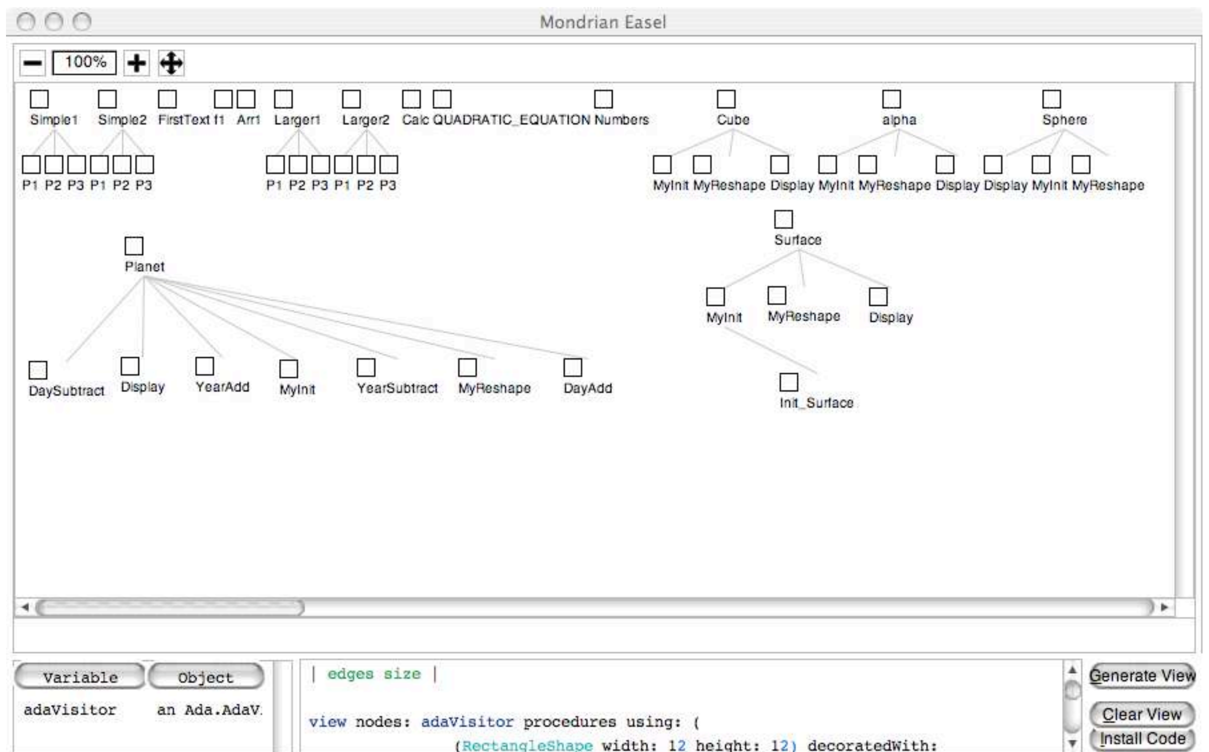
We are using the visitor pattern to collect the whole parse tree. By visiting the tree different Collections of program elements are created. Especially procedures and functions which can be used to build up a moose model:

```
procedures do: [:each | | function |
function := FAMIXFunction new.
function setName: each name.
model addEntity: function.
].
functions do: [:each | | function |
function := FAMIXFunction new.
function setName: each name.
model addEntity: function.
```

```
].
^model
```

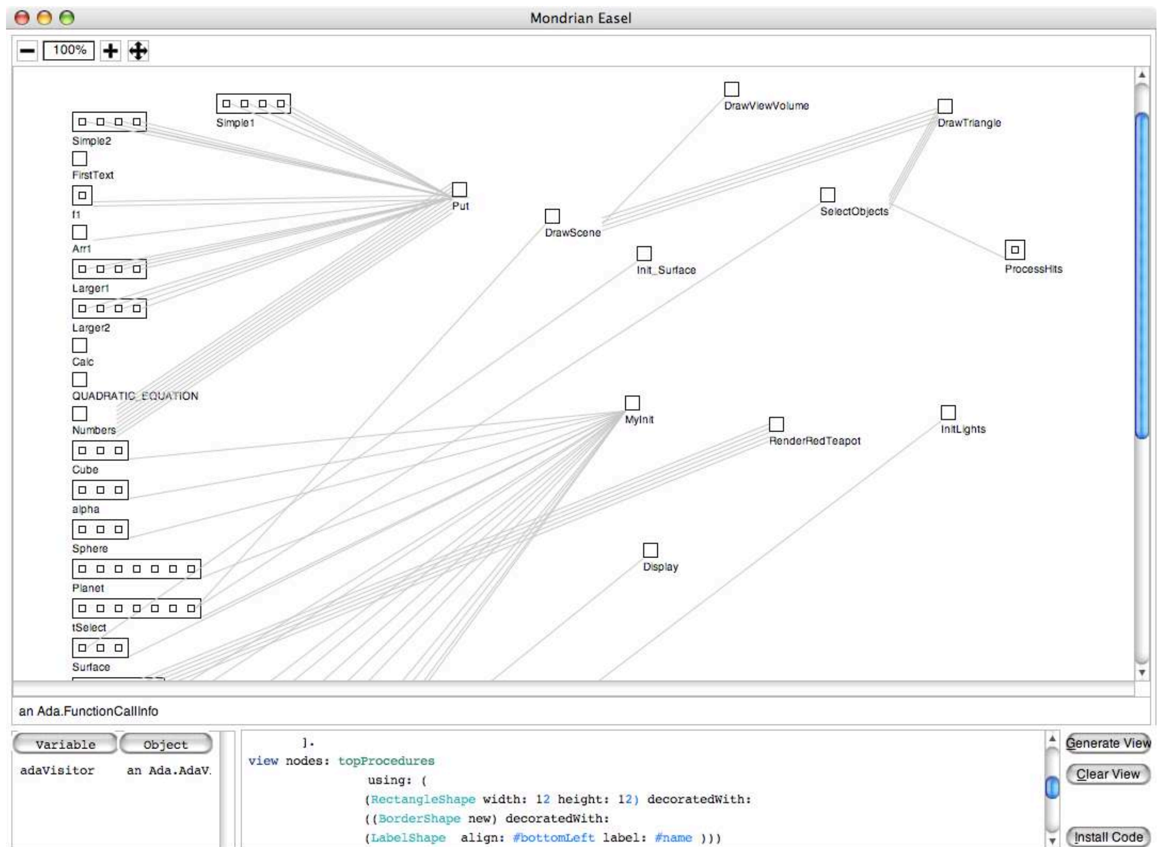
5.1 Exporting to Mondrian

By adding some code to visitor-method of the Procedure- and Function-Classes we may capture for every Ada-Procedure the containing informations: in which procedure eventually this procedure is contained and which procedures this procedure containes. Most of the parsed Ada-code can be found also on the internet [1].



Containment Hierarchy

Simillary we create a class with all procedure-/function-calls with their origin and target. Using again a small Mondrian-Script we get a visualitation of both the containment and the calls.



Containment Information and Calls displayed as lines between Procedures.

6 Problems, Troubleshooting and Testing

6.1 Shift/Reduce conflicts

The parser generator cannot decide whether to perform a shift or a reduce. Possibly the grammar rule complete is not complete. An example where such a conflict can happen is the dangling else:

if ... then ...
if ... then ... else ...

so in `if a then if b then c else d` to which if does the else belong?

6.2 Reduce/Reduce conflicts

This means the parser generator has found an ambiguity in the grammar definition. I.e there is more than one way to parse a single word into a sequence. One should try to solve this by rewriting the definitions so the ambiguity disapears. Sometimes this conflicts are quite difficult to find. Then the parser could try to use the first possible reduce.

Sample of a small grammar with a reduce/reduce conflict:

```
Start : A | B;
A : "z" ;
B : "z" ;
```

6.3 Other Problems and Troubleshooting

- Integrating Context Clauses in a way so no reduce/reduce conflicts are added was difficult.
- Parsing comments: Since comments can stand everywhere in the source text it is not so clear how to integrate them into the definitions. I wrote a Smalltalk function which removes comments out of source texts before they get parsed.

6.4 Testing

We used the generated parsers to test if they can parse different sources without errors. Usually they produce a “token not expected” error when they can’t apply any rule at a specific input text passage.

To test we used the following Smalltalk program text (among other commands) to pass a text to a specific parser (usually the StandardParser):

tree := StandardParser parse: (LoadAdaFiles new preprocessAdaString: AdaTestCode adaCode).

self assert: tree notNil.

self assert: (tree isKindOf: Program).

Since we used the visitor pattern we could add:

```
visitor := AdaVisitor new.
tree acceptVisitor: visitor.
self assert: visitor procedures size = 4.
self assert: visitor contextClauses size = 4
```

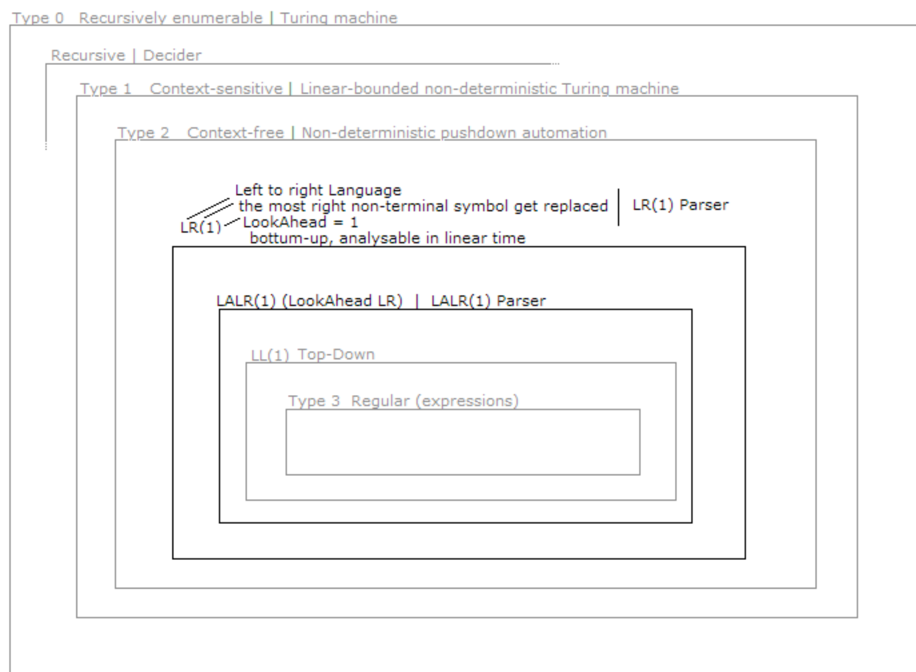
A Appendix section

A.1 About the different coverages of parsers — from arbitary texts to texts with special structure

In general one could say that we want to recognize the structure of an arbitary text(-string). However, programming languages — which we cover here — have to be con-

formant to some much more tightening rules. Typically they are defined in a recursive descent way. This means their definition is given a list which can be traversed in different ways from top to bottom. Every different path stands for a different (set of) valid source texts.

The Chomsky hierarchy [3] orders different types of languages accordingly to the means one needs to recognize their structure and content.



The Comsky hierarchy. On the left the language type and on the right what kind of program (or machine) is needed to parse it (recognize its structure). To parse computer program texts one usually uses a LR or an LL-parser. SmaCC is a LR-parser(-generator) which can parse more than a LL-parser but arising problems can be harder than those of such parsers.

B Small examples

The synatx used in SmaCC is very similar to the Backus-Naur form (BNF) which can be used to express context-free grammars or subsets of them (see figure above). There are two types of words: non-terminals and terminals. Non-terminals help to describe the syntax, terminals are words which may occur literally in the parsable text and are “quoted”. On the left side stands a non-terminal symbol which is defined by the right side. The following table shows the most important possibilities to compose the right side.

<table>
<tr><th>symbol</th><th>description</th><th>sample usage</th></tr>
<tr><td>“word”</td><td>this word occurs literally in the parsable text</td><td>if: “if”;</td></tr>
<tr><td><word></td><td>a ’word’ defined in the Scanner</td><td>function: “function” <identifier>;</td></tr>
<tr><td>|</td><td>Choice (notice, that no parantheses are needed for “and” then”)</td><td>operator: “and” | “or” | “xor” | “and” “then”;</td></tr>
<tr><td>[.] or (.)?</td><td>may occur not or one time (the part in parantheses is optional)</td><td>factor: primary ["**" primary];</td></tr>
<tr><td>(.)*</td><td>may occur not or as many times as wanted</td><td>term: factor (multiplying_operator factor)*;</td></tr>
<tr><td>word</td><td>non-terminal, has to be defined one time standing on the left side</td><td></td></tr>
</table>

B.1 An Example: Expressions

We would like to parse expressions like (30*heigth > 4+4^4) or not(A=>B) which may occur in a source code. Below is a set of definitions we would enter in SmaCC as parser-definitions and which would help it to parse such expressions.

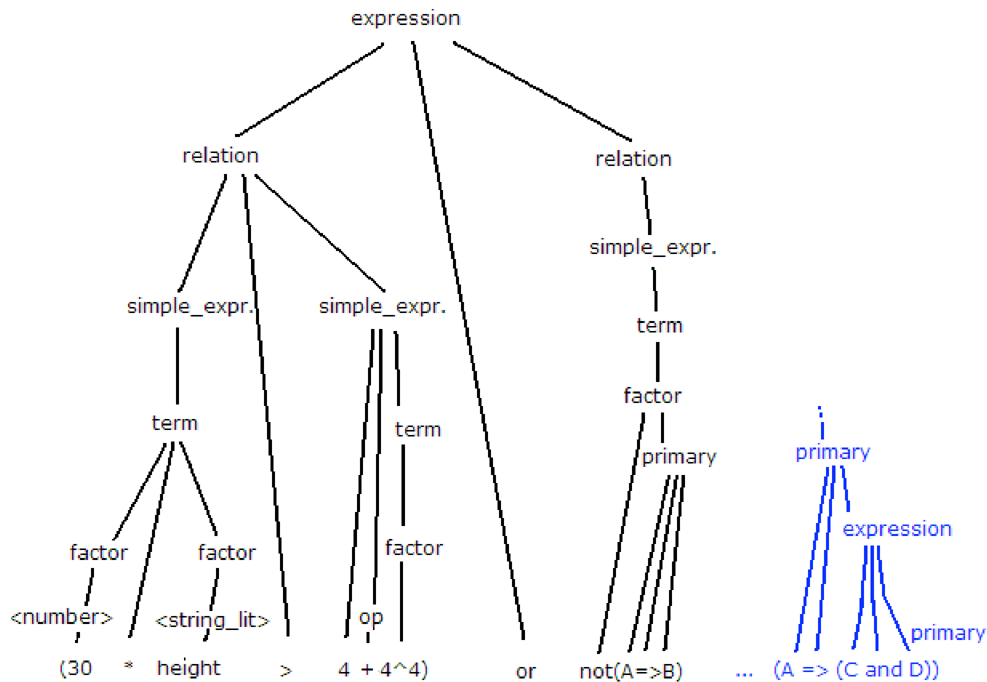
<table>
<tr><td>expression:</td><td>relation(("and" | "and then" | "or" | "or else" | "xor") relation)*;</td></tr>
<tr><td>relation:</td><td>simple_expression rel_op simple_expression;</td></tr>
<tr><td>simple_expression:</td><td>unary_adding_op? term (binary_adding_op term)*;</td></tr>
<tr><td>term:</td><td>factor (multiplying_op factor)*;</td></tr>
<tr><td>factor:</td><td>primary ("^" primary)? | "abs" primary | "not" primary | ":" primary | "=>" primary;</td></tr>
<tr><td>primary:</td><td><string_lit> | <number> | inner_function_call | "("expression")";</td></tr>
</table>

Notice that non-terminals (definitions) standing above others relate to definitions below. But this does not have to be always the case. For example primary on the bottom relates to the first definition expression. We will see that also recursive self-definitions are possible.

Remark on inner_function_call: It is named ’inner’ because in Ada common function calls are followed by a ’;’ as they can be fully statements but not within expressions.

The Parser-Generator (SmaCC) will generate (compile) a parser which can analyse the input-string (our expression) and output the syntax-tree of it, given the input is a valid expression (conforming to its definition).

B.2 SyntaxTree of the example



Syntax-Tree of the example-expresssion. We could replace B with (C and D) to have an example (blue) of an expression in an expression. So the tree has not to go strictly from expression (on top) downwards (visiting the definitions below) but also can again 'go back' to definitions earlier in the list. But at the bottom there have to be always terminals (in this case primaries or operators).

References

[1] Ada code parsed (opengl program). http://www.adapower.com/index.php? Command=Class&ClassID=AdaGraphics&Title=Graphics+Examples.

[2] Definition of ada 83. http://www.adahome.com/LRM/83/RM/rm83html.

[3] Visual works. http://www.cincomsmalltalk.com.

[4] John Brant and Don Roberts. SmaCC, a Smalltalk Compiler-Compiler. http://www.refactory.com/Software/SmaCC/.

[5] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.