Description of CST->AST transformations in SableCC 3

Introduction

What has been changed?

- Addition of a new section called Abstract Syntax Tree.
- New syntax for transformations specification.
- SableCC3 provides support for SableCC2 grammars.

To be able to benefit of AST transformations in SableCC, it is necessary to add a new *Abstract Syntax Tree* section's to the grammar.

This section must be placed after the Productions section's. It contains the grammar of the Abstract Syntax Tree, so all the Nodes in this tree are instances of classes generated using this part of the grammar. In this new section, the syntax of productions(ast_prod) is the same as in the section *Productions* of SableCC2 grammars. All the classes used to represent this tree are generated using the productions specified in this new section.

A typical grammar of SableCC3 should contains these following sections:

- Helpers
- Tokens
- Ignored Tokens
- Productions, and
- Abstract Syntax Tree.

All these sections as it is specified in the SableCC3's grammar are optional.

Transformations

They are specified in the section Productions. Two intrinsically bound transformation's actions are needed to perform :

- the first is a for the production itself
- and the second is for alternatives of this production

Transformation of the production

A production is transformed in one or several AST productions or tokens.

Such a transformation looks like:

```
production {-> prod_transform1 prod_transform2* ... prod_transformN} =
      element1 element2 ...  elementN
      {-> alt_transform1 alt_transform2 ... alt_transformN}
```

Where prod_transformation1, prod_transformation2, ..., prod_transformationN are AST productions or Tokens. The existence of the operator * indicates that it is a list of this element and not just the element itself.

During the parsing time of this mini grammar's compliant program, at the reduction phase of the alternative below, instead of constructing a traditional production node, the parser will constrcut the following items :

- a node of type prod_transform1,
- a homogeneous list containing elements of type prod_transform2,
- a node of type prod_transform3
- ...(a type prod_transformi node or list of type prod_transformi node depending on the presence of * operator or not.)
- and finally a node of type prod_transformN.

Transformation of alternatives

(Note: In order to use SableCC grammar terminology, we will refer to: *element* for transformations of one production and *term* for tranformations of an alternative.)

The transformation of an alternative is guided by the transformation of the production :

- the number of terms and the number of elements should be the same.

- the type of these terms shoud also correspond to the type of elements of the production.

If we look at our example of production in the paragraph 2.1, it means that:

- prod_transform1 should be the same type that alt_transform1,
- prod_transform2 should be the same type that alt_transform2 and so forth for all another ones until prod_transformN.

(Note: When we say that prod_transform1 is the same type as alt_transform1, that means : alt_transform1 are one of the alternatives of the production prod_transform1 or alt_transform1 and prod_transform1 can match the same token).

There are four types of terms for the transformation of an alternative:

1. Getting an already existing element :: (ident)
2. New alternative (New production[.nameofalternative]) :: creation of a new node
3. List creation ([elem1 elem2 ...]) :: creation of a homogeneous list of terms
4. Elimination (Null) :: used in general to replace an element or to eliminate the effect of another one.
5. Empty transformation :: used in general to get ride of all the the subtree

In order to describe with more precise manner these terms, let use one example of SableCC3 grammar with transformations (this example can be found in appendix of this document).

1. Getting an already existing element

```
(exp_list_tail {-> exp} = comma exp {-> exp};)
```

In the production *exp_list_tail*, we have an alternative with two elements: *comma* and *exp*. In the transformation of this alternative, we only keeps the element exp. Here, we are just taking an already existing element of the tree.
Notice that the production exp_list_tail is supposed to change to exp, and the term of the transformation of alternative is also an exp. Therefore the concordance of type is respected.

In the production *factor*, if we look at the grammar, we realize that *term* is itself a production that is transformed to *exp*. It means that in our tree, all the *term* are transformed to *exp*.

Hence, "term.exp" refers to the element "exp" which already replace production *term*; so term.exp is not an element of type *term* but an element of type *exp*. That makes us once again respect the required concordance of type.

2. New alternative

```
( exp = {plus} exp plus factor {-> New exp.plus(exp, factor.exp)};  )
```

The syntax is: *New* following by the appropriate name the alternative of the production. If the alternative carries an explicitly specified name between like nameofalternative, the syntax must be: *New nameofproduction.nameofalternative (parameters)*, otherwise *New nameofproduction (parameters)*. In this case, nameofproduction and nameofalternatives are similar.

nameofproduction must be the name of a definite production in the AST section. And parameters must be describe as:

```
parameter1, parameter2, parameter3,... .
```

In the case of the transformation 2(see Appendix), exp.plus refers to the alternative *{plus} [l]:exp [r]:exp* of production *exp* that state in the Abstract Syntax Tree section. This alternative is composed of two elements which types are *exp*. It is why for parameters of *New exp.plus()*, we have *exp* and *factor.exp* which are two elements of type exp.

3. List creation

```
(exp_list {-> exp*} = exp exp_list_tail {-> [exp_list_tail.exp exp] };)
```

To construct a list elements, the syntax used is *(elem1 elem2 elem3... elemN)*, where elem1 ... elemN are all elements of the same type.

In production *exp_list*, *(exp_list_tail.exp exp)* is a list of exp. exp_list_tail.exp represents an exp type's element; because exp_list_tail is transformed to exp.

4. Elimination

```
(Null)
```

In the grammar in appendix, there is no such transformation. To make an illustration, we can modify one of alternatives. For example we can transform the production *exp* of section Productions to :

```
   exp = {plus} exp  plus factor  {-> New exp.plus(exp, factor.exp)} |
         {minus} exp minus factor {-> New exp.minus(exp, Null)}      |
             {factor} factor      {-> factor.exp };
```

It means that we don't keep the term factor anymore in the alternative minus. Null is an element that is compatible with all types except lists. So it can be used everywhere an element is needed. If a empty list is needed, just used this : ().

Empty transformation

```
exp_list_tail {-> } = comma exp {-> };
```

There is a difference between empty and Null transformation. in the case of null transformation (exp_list_tail {-> exp} = comma exp {-> Null}), the corresponding node can still be accessed by writing *exp_list_tail.exp* even if the associated node contains null reference. That is *exp_list_tail.exp* is an expression type element but it contains null reference.

But in the case of empty transformation, one just get rid of the production. exp_list_tail cannot be accessed anymore.

Implicit transformations

When transformation is not specified in the grammar, an implicit transformation is introduced by the parser either for productions and alternatives. Example: A production like

- production = elem1 elem2 * elem3+ elem4?; is transformed to
- production {-> production} = elem1 elem2 * elem3+ elem4? {-> New production(elem1, [elem2], [elem3], elem4) };

This implicit kind of transformation is always done for all productions and alternatives with no explicit transformations.

Restrictions

1.

   *** For the specification of transformations, the first production of Productions' sections should be transformed to the first production of the AST section. In our example(see appendix), we should have :

      grammar {-> grammar} , what is seen to be the case because

grammar = elems ... is transformed in
grammar {-> grammar} = elems ... by the parser.

2.

*** In transformations of alternative, an element with an operator * or + can only be referred to in a list transformation. For example :
prod {-> elem} = elem1 elem* {-> elem }; is not correct even if the concordance type is still respected.
It should rather be :
prod {-> elem*} = elem1 elem* {-> (elem) };

Appendix

```
Package expression;

Helpers

    digit = ['0' .. '9'];
    tab = 9;
    cr = 13;
    lf = 10;
    eol = cr lf | cr | lf; // This takes care of different platforms

    blank = (' ' | tab | eol)+;

Tokens
    l_par = '(';
    r_par = ')';
    plus = '+';
    minus = '-';
    mult = '*';
    div = '/';
    comma = ',';

    blank = blank;
    number = digit+;


Ignored Tokens
```

```
    blank;

Productions

    grammar = exp_list {-> New grammar([exp_list.exp])};

    exp_list {-> exp*} = exp exp_list_tail* {-> [exp exp_list_tail.exp]};

    exp_list_tail {-> exp} = comma exp {-> exp};

    exp =
          {plus}   exp plus factor  {-> New exp.plus(exp, factor.exp)  }
        | {minus}  exp minus factor {-> New exp.minus(exp, factor.exp) }
        | {factor} factor           {-> factor.exp}
    ;

    factor {-> exp} =
          {mult} factor mult term {-> New exp.mult(factor.exp, term.exp )}
        | {div}  factor div term  {-> New exp.div(factor.exp, term.exp ) }
        | {term} term             {-> term.exp}
    ;

    term {-> exp} =
          {number} number          {-> New exp.number(number)}
        | {exp}    l_par exp r_par {-> exp}
    ;


Abstract Syntax Tree

    grammar = exp* ;

    exp = {plus}   [l]:exp [r]:exp
    | {minus}  [l]:exp [r]:exp
    | {div}    [l]:exp [r]:exp
    | {mult}   [l]:exp [r]:exp
    | {number} number
    ;
```

Agbakpem Komivi
5/23/2003