

Generic_First_Follow Modification Notes

- Operations provided by `Generic_First_Follow` must not assume that all literals of type `Symbol` are terminals (`Terminal_First..Terminal_Last`) or nonterminals (`Productions'Range`). Other literals of type `Symbol`—e.g., semantic rule symbols—might exist and should be effectively ignored when computing first and follow sets.

Generic_CFG_to_NFA Modification Notes

- Semantic rule symbols might occur together with grammar symbols on the right-hand side of a production. The NFA constructed herein should make no transition from an LR(0) item in which the symbol following the dot is a semantic rule symbol. The leftmost semantic rule symbol and any following symbols should be effectively ignored in this construction.

Generic_SLR Modification Notes

- The type `Handle` should be modified to include a `Dot` component which indicates the position of the dot in an LR(0) item.
- `Reduce_By (I, Lookahead)` should return a `Handle` $A \rightarrow \alpha_1 \bullet \alpha_2$ such that $A \rightarrow \alpha_1 \bullet \alpha_2 \in I$ and `Lookahead` $\in \text{Follow}(A)$ and α_2 is empty or begins with a semantic rule symbol. If no such `Handle` exists, `No_Handle` is raised.

Generic_Shift_Reduce Modification Notes

```

generic
-----
-- T y p e s
--
type Symbol          is ( <> )
type Phrase          is array ( Positive range <> ) of Symbol
type To_Phrase       is access      Phrase
type Phrases         is array ( Positive range <> ) of To_Phrase
type To_Phrases      is access      Phrases
type Token ( X : Symbol ) is private
type To-Token        is access      Token
type Tokens          is array ( Positive range <> ) of To-Token
type Context_Free_Grammar is array ( Symbol range <> ) of To_Phrases
-----

-- C o n s t a n t s
--
Terminal_First : Symbol
Terminal_Last  : Symbol
Productions    : Context_Free_Grammar
Start          : Symbol := Productions'First
Follow_Start   : Symbol := Terminal_Last
-----

-- O p e r a t i o n s
--
with function Lookahead return Symbol          is <> ;
with function Lexeme    return String         is <> ;
with procedure Evaluate ( a : in out Token ; Lexeme : String ) is <> ;
with procedure Evaluate ( j : Positive ; A : in out Token ; Alpha : Tokens ) is <> ;

procedure Generic_Shift_Reduce;

```

- `Generic_Shift_Reduce` should evaluate attributes of an S-attributed SDD while constructing an annotated parse tree from the bottom up.
- The discriminant record type `Token` and the types `To-Token` and `Tokens` have been added to the generic formal types. An object of type `Token` contains a grammar symbol and its associated attributes.

- When reducing using the handle $A \rightarrow \alpha_1 \bullet \alpha_2$, apply all semantic rules in α_2 to compute the values of synthesized attributes associated with the nonterminal A depending on attributes associated with grammar symbols in α_1 .
- **Evaluate** (a , **Lexeme**) computes the value of a synthesized terminal attribute in the token a depending on **Lexeme**.
- **Evaluate** (j , A , **Alpha**) computes the value of a synthesized nonterminal attribute in the token A using semantic rule **Alpha**(j) and depending on the attributes in tokens **Alpha**(1), **Alpha**(2), ..., **Alpha**($j-1$).

```

package Lisp_CFG is

  type Symbol is (
    T, NIL, QUOTE, CAR, CDR, CONS, EQ, ATOM_t, COND, LAMBDA, LABEL,
    '(' , ')', id, space, comment, end_of_source,

    expr1, expr, atom_nt, list, cases, exprs, anonymous_function,
    named_function, case_nt, formals,

    interpret, identity, list_zero, list_zero_plus, list_one_plus,
    list_two, list_three, list_prepend, Churchism );

-----
-- Lexical Analyzer
--
subtype Terminal is Symbol range T..end_of_source ;

function Lookahead return Terminal ;
function Lexeme      return String  ;

No_Lexeme_Prefix : exception ;

-----
-- Syntax Specification
--
type Phrase          is array ( Positive range <> ) of Symbol      ;
type To_Phrase       is access                               Phrase   ;
type Phrases         is array ( Positive range <> ) of To_Phrase   ;
type To_Phrases      is access                               Phrases  ;
type Context_Free_Grammar is array ( Symbol range <> ) of To_Phrases ;
--
Productions: constant Context_Free_Grammar := (

  expr1          => new Phrases'( 1 => new Phrase'(
    expr, interpret
  )),
  expr          => new Phrases'(      new Phrase'(
    atom_nt, identity
    list, identity
  )),
  atom_nt       => new Phrases'(      new Phrase'(
    T, identity
    id, identity
  )),
  list          => new Phrases'(      new Phrase'(
    NIL, identity
    '(' , QUOTE, expr, ')', list_two
    '(' , CAR, expr, ')', list_two
    '(' , CDR, expr, ')', list_two
    '(' , CONS, expr, expr, ')', list_three
    '(' , EQ, expr, expr, ')', list_three
    '(' , ATOM_t, expr, ')', list_two
    '(' , COND, cases, ')', list_one_plus
    '(' , exprs, ')', list_zero_plus
    '(' , anonymous_function, exprs, ')', list_one_plus
    '(' , named_function, exprs, ')', list_one_plus
  )),
  cases         => new Phrases'(      new Phrase'( 1 =>
    list_zero
    case_nt,cases, list_prepend
  )),
  exprs         => new Phrases'(      new Phrase'( 1 =>
    list_zero
    expr,exprs, list_prepend
  )),
  anonymous_function => new Phrases'( 1 => new Phrase'(
    '(' ,LAMBDA,'(',formals,')',expr,')', Churchism
  )),
  named_function  => new Phrases'( 1 => new Phrase'(
    '(' ,LABEL,id,anonymous_function,')', list_three
  )),
  case_nt        => new Phrases'( 1 => new Phrase'(
    '(' ,expr,expr,')', list_two
  )),
  formals        => new Phrases'(      new Phrase'( 1 =>
    list_zero
    id,formals, list_prepend
  )),
  )));

end Lisp_CFG;

```

- Semantic rule symbols have been added to grammar symbols in the type Symbol.
- Semantic rule symbols have been added to the right-hand sides of productions to indicate which semantic rules are associated with a production.

```

with Lisp_CFG;
use Lisp_CFG;

package Lisp_SDD is

-----
-- S e m a n t i c   A t t r i b u t e s
--
type Node_Type          is ( atom_node, pair_node ) ;
type To_String          is access String           ;
type Node ( Atomicity : Node_Type )              ;
type Tree              is access Node            ;
type Node ( Atomicity : Node_Type ) is

    record
        case Atomicity is
            when atom_node => Id      : To_String ;
            when pair_node => Car, Cdr : Tree   ;
        end case;
    end record;

type Token ( X : Symbol ) is

    record
        case X is
            when '(' | ')' | space | comment | end_of_source |
                interpret..Churchism => null      ;
            when others                => Attribute : Tree ;
        end case;
    end record;

type To-Token is access Token ;
type Tokens   is array ( Positive range <> ) of To-Token ;

-----
-- S e m a n t i c   R u l e s
--
procedure Evaluate (          a : in out Token ; Lexeme : String ) ;
procedure Evaluate ( j : Positive ; A : in out Token ; Alpha : Tokens ) ;
--
Semantic_Error : exception ;

end Lisp_SDD;

```

Attribute Terminology

A *Tree* is *null* if it is the null pointer. A *Tree* is an *atom* if it is a pointer to a node whose *Atomicity* is *atom_node*; the *Id* of such a node is called an *identifier*. A *Tree* is a *pair* if it is a pointer to a node whose *Atomicity* is *pair_node*; the *Car* and *Cdr* of such a node are called *components*. A *Tree* is a *list* if it is null or it is a pair whose *Cdr* component is a list. The *Car* component of such a node in a list is called an *element*. A *binding* is a pair whose *Car* component is an atom having a non-reserved identifier. The *Cdr* component is called a *value*. An *environment* is a list whose elements are bindings.

- `Evaluate (a, Lexeme)` computes the value of `a.Attribute` depending on `Lexeme`. If `a.Symbol` is `T`, `QUOTE`, `CAR`, `CDR`, `CONS`, `EQ`, `ATOM_t`, `COND`, `LAMBDA`, `LABEL`, or `id`, `a.Attribute` should be assigned an atom tree whose identifier is uppercase `Lexeme`. If `a.Symbol` is `NIL`, `a.Attribute` should be assigned a null tree.
- `Evaluate (j, A, Alpha)` computes the value of `A.Attribute` using semantic rule `Alpha(j).Symbol`. The semantic rules are defined in the Project SDD handout and refer directly or indirectly to the

following semantic functions:

```
function Leaf      ( S          : String ) return Tree ;
function Cons     ( Car, Cdr    : Tree   ) return Tree ;
function Atom     ( T          : Tree   ) return Boolean ;
function Pair     ( T          : Tree   ) return Boolean ;
function Car      ( T          : Tree   ) return Tree ;
function Cdr      ( T          : Tree   ) return Tree ;
function Cadr     ( T          : Tree   ) return Tree ;
function Caddr    ( T          : Tree   ) return Tree ;
function Caar     ( T          : Tree   ) return Tree ;
function Cadar    ( T          : Tree   ) return Tree ;
function Caddar   ( T          : Tree   ) return Tree ;
function Eq       ( S, T       : Tree   ) return Boolean ;
function Case_Value ( Cases, Global : Tree ) return Tree ;
function Binding  ( Id, Global   : Tree ) return Tree ;
function Append   ( Local, Global : Tree ) return Tree ;
function Local    ( Formals, Values : Tree ) return Tree ;
function Values   ( Actuals, Global : Tree ) return Tree ;
function Value    ( T, Global     : Tree ) return Tree ;
procedure Put     ( T            : Tree ) ;
```

1. `Leaf` returns an atom whose identifier is `S`.
2. `Cons` returns a pair whose components are `Car` and `Cdr`.
3. `Atom` returns true if and only if `T` is an atom.
4. `Pair` returns true if and only if `T` is a pair.
5. `Car` and `Cdr` return the `Car` and `Cdr` components of the pair `T`. If `T` is not a pair, a *semantic error* occurs.
6. `Cadr(T)` is `Car(Cdr(T))`. `Caddr`, `Caar`, `Cadar`, `Caddar` are similarly defined.
7. `Eq` returns true if and only if `S` and `T` are identical trees.
8. `Case_Value` returns the value of `Cadr(Case)` in the environment `Global`, where `Case` is the first element of `Cases` such that the value of `Car(Case)` in the environment `Global` is not null. If no such element exists, a *semantic error* occurs.
9. `Binding` returns the first binding in the environment `Global` whose `Car` component identifier is `Id`. If no such element exists, a *semantic error* occurs.
10. `Append` returns the environment whose bindings are the bindings of the environment `Local` followed by the bindings of the environment `Global`.
11. `Local` returns the list of bindings obtained by pairing successive elements from the lists `Formals` and `Values`.
12. `Values` returns the list whose elements are the values in the environment `Global` of the respective elements in `Actuals`.

13. **Value** returns “the value of” T “in the environment” Global, as defined by the following pseudo-code:

```

if T is null then
    return T
else if T is an atom then Case T.Id of
    T:      return T
    others: return Cdr (Binding(T, Global))
else if Car(T) is an atom then Case Car(T).Id of
    QUOTE:  return Cadr (T)
    CAR:    return Car  (Value (Cadr(T), Global))
    CDR:    return Cdr  (Value (Cadr(T), Global))
    CONS:   return Cons (Value (Cadr(T), Global),
                        Value (Caddr(T), Global))
    EQ:     if      Eq   (Value (Cadr(T), Global),
                        Value (Caddr(T), Global)) then return Leaf(T)
                        else return null
    ATOM:   if      Atom (Value (Cadr(T), Global)) then return Leaf(T)
                        else return null
    COND:   return Case_Value (Cdr(T), Global)
    others: return Value (Cons (Cdr(Binding(Car(T), Global)),
                                Cdr(T)),
                            Global)
else if Caar(T) is an atom then Case Caar(T).Id of
    LAMBDA: return Value (Caddar(T),
                        Append(Local(Cadar(T),
                                    Values(Cdr(T), Global)),
                                Global))
    LABEL:  return Value (Cons(Caddar(T), Cdr(T)),
                        Cons(Cons(Cadar(T), Caddar(T)),
                                Global))
    others: semantic error
else
    semantic error

```

14. **Put** prints an expression (expr) which has an attribute (expr#attr) equal to T.