

13. Macros

e is a truly extensible language. It can be extended all the way down to its syntax and lexicon using macros — declared by *define-as* statements. Unlike macros in languages such as C, which are preprocessor based, *e* macros are real syntactic rules. They actually modify the grammar of the language — introducing new derivation rules. Macros can be used not only to overload existing operators and constructs, but also to add whole new construct with new keywords to the language. The newly declared constructs are implemented by defining a syntactic expansion, reducing them to existing constructs.

13.1 Syntax overview

e syntax in general, and user-defined syntax (introduced by macros) in particular, is specified in a way similar to context-free grammars. Each macro can be viewed as defining an alternative for a non-terminal production rule. But there are a number of essential differences between context-free grammars and the *e* analogue. Here are the important ones:

- The terminals of the grammar are simply characters from the input (after trivial normalization and preprocessing). There are no separate lexical rules.
- The term-sequence in production rules can be expressed using regular-expression-like operators, such as optional and alternative subsequences.
- All ambiguities are eliminated by a set priority order over all production rules.

These principles make the introduction of new syntax more flexible and intuitive than it would be using standard parser-generator schemes.

13.2 Defining a macro

Purpose	Define a new construct by reduction
Category	Statement
Syntax	define <tag> <i>non-terminal-type</i> <"match-string"> as { <i>replacement</i> }
Parameters	<i>tag</i> An identifier that, together with the <i>non-terminal</i> , is used as a name for the macro (and thus shall be unique). It plays no actual role in the definition.
	<i>non-terminal-type</i> Any one of <i>statement</i> , <i>struct_member</i> , <i>action</i> , or <i>exp</i> , which are non-terminals in <i>e</i> grammar that stand for the corresponding **four?? syntactic categories (see 2.2). This parameter indicates which of them is being extended (overloaded) by the macro.
	<i>match-string</i> The actual match expression enclosed between quotes (" "). A <i>match expression</i> is a sequence of terminals and non-terminals that constitute a new choice for the derivation of the specified <i>non-terminal</i> (see 13.3).
	<i>replacement</i> The parameterized replacement code to which the new construct is reduced (see 13.5).

A *macro* extends the language, by declaring a new construct under one of the exiting syntactic categories. The meaning of the construct is given in terms of a transformation rule (expansion) to legal *e* code, with the actual substrings parsed from the input for sub-non-terminals serving as parameters. The macro is expanded whenever the match expression applies to the input that is being parsed under the requested category.

Syntax example: ****Update??**

```
define <largest'action> "largest <exp> <num>" as {
  if <num> > <exp> then {<exp> = <num>};
};
```

13.3 Match expression structure

The *match expression* consists of a sequence of terminals and non-terminals, possibly with regular expression operators such as alternative and optional sub-sequences.

13.3.1 Match expression terms

Terminals in the *e* grammar are simply ASCII characters. The grammar does not presuppose independent lexical analysis. The terminal part of a production is given literally in the match expression string. Some characters have special meaning in a match expression (see 13.3.2), so they need to be escaped (\) to be taken as terminals.

The non-terminal types for user macros are the same ones available for extension: *statement*, *struct_member*, *action*, and *exp* which stand for a construct of the corresponding syntactic category (see 2.2). A few “auxiliary” non-terminal types are also available: *name*, *num*, *Type*, *file*, *block*, and *any*. An occurrence of a non-terminal in the match expression is marked by enclosing it with < and >. It consists of a non-terminal type selector, optionally preceded by an identifier serving as a tag (the format is <[tag']non-terminal-type>). This constitutes a declaration of a formal syntactic parameter that can be used in the replacement. For example, the occurrence of <left'exp> inside a match expression is a declaration of a parameter by that name of the *exp* non-terminal type.

The role of the auxiliary non-terminals is shown in Table 1. *See also*: 2.1.

Table 1—Auxiliary non-terminals

Parsing element	Description
<name>	A legal <i>e</i> name.
<num>	A literal numeric constants.
<Type>	Any legal type name, simple or compound.
<file>	A UNIX-style file name. **Only used with UNIX files??
<block>	A series of actions delimited by ; and enclosed between { }.
<any>	Any (possibly empty) sequence of characters from the input.

13.3.2 Literal characters and escaping

The characters [,], (,), and | are used as regular-expression operators (option, grouping, and alternation, respectively). These, together with < and >, are not taken literally inside a match expression, unless they are escaped (\).

13.3.3 Meta-grammar of match expression

The following grammar (usable in **LALR** parsers) defines the syntax of match expressions described informally above (non-literal terminals are in italics).

```

e_match_expression ::= sequence
                    | e_match_expression | sequence
sequence ::= e
           | sequence term
           | sequence grouping
           | sequence option
grouping ::= ( e_match_expression )
option ::= [ e_match_expression ]
term ::= literal
       | non_terminal
literal ::= character
         | literal character
non_terminal ::= nt_selector
              | identifier * nt_selector
nt_selector ::= statement | struct_member | action | exp | type | name | num
              | block | file | any

```

identifier is the standard *e* identifier and *character* is any ASCII character except the special ones: [,], (,), |, <, >, and \, or any one of these special characters when preceded by \.

13.3.4 Proto-syntax

A number of characters have a fixed syntactic function in *e*. They are [,], (,), {, }, and ", all of which signify subordination of one construct to another. This syntactic role is reflected in the structure of macro match expression. These characters cannot be placed just anywhere in the match expression of a macro; they need to be balanced, enclosing only a single non-terminal. This restriction can be expressed by adding the following rules to the grammar in 13.3.3.

```

term ::= literal
      | non_terminal
      | bracketed_non_terminal
bracketed_non_terminal ::= \ ( non_terminal \ )
                       | \ [ non_terminal \ ]
                       | { non_terminal }
                       | " non_terminal "

```

Parentheses (()) and square brackets ([]) need to be escaped (\) in the match expression to be taken literally, i.e., to represent these same characters in the input stream. Otherwise, they are taken as regular-operators. With this addition, the set of literal terminals shall be excluded from the *character* terminal specified in the grammar in 13.3.3.

13.4 Interpretation of match expressions

This section defines how interpretation of match expressions occurs.

13.4.1 Priority on production choices

A macro associates a new match choice with the non-terminal that is being extended. Each non-terminal already has a list of built-in production choices associated with it and might further have any number of production choices defined by previous macros. This does not limit the new match expression in any way and no ambiguity can be introduced thereby. The reason is that, unlike context-free grammars, the productions are prioritized. Match expressions are tested according to their priority — the first one that succeeds gives the correct derivation for the input.

The priority is determined by the definition order of macros in the code — those whose definition appears later get higher priority (see [Clause xx, analysis order](#)). Thus, production choices declared by macros always have higher priority than built-in productions. In general, existing (built-in or user-define) syntax can be overwritten by a new macro.

13.4.2 Recursive-decent interpretation

The modification to the grammar introduced by macros is best understood in terms of a recursive-decent backtracking parsing algorithm. Each non-terminal represents a routine which takes the string to be parsed as input and either succeeds in consuming some prefix of it or fails to do so. A non-terminal routine *succeeds* if at least one of the match expressions associated with it succeeds. The production that is actually used in this case is the production of highest priority that consumes the whole input. A match expression applies to the input if both its terminal parts match it and the substrings left for non-terminals succeed to be consumed by the corresponding non-terminal routine. The non-terminal *fails* if no match expression associated with it matches the actual input string. In this case, the algorithm backtracks further.

The transformation rule declared by the macro is activated whenever the production has matched some section of the input. The code generated by the transformation is normalized, preprocessed, and tested again against productions of the original non-terminal. This [can fail](#) if the transformation rule itself generates code that is not well formed. In this case, backtracking proceeds further up (and could result in a syntax error).

13.5 Replacement code

The expansion rule declared by the macro is given as parameterized (template) code segment. The replacement segment is taken literally, except for occurrences of the syntactic parameters enclosed by `< >`. These can be of [three](#) kinds:

- a) Non-terminal parameters as in the `<[tag']non-terminal-type>` format — referring to the corresponding non-terminal in the match expression (if there is more than one non-terminal of the same type in the match expression with no unique tag, the reference is undefined).
- b) Non-terminal parameters with a default value in the format: `<[tag']non-terminal-type|val>` (*val* being any text) — either referring to the corresponding non-terminal, as above, or replaced with *val* [when](#) the non-terminal appears in the match string as optional or alternative, and has not actually been matched.
- c) Implicit submatch parameters in the format `<n>` (*n* being a positive integer) — referring to the *n*th submatch of the matched input. Submatches are the sections of input string matched either inside the grouping operator or by sub-non-terminals, starting from left to right, the first of which is 1.

The code that is generated by instantiating the template with the actual syntactic parameters is parsed again. It shall be a legal construct of the same category as the one being replaced or a series thereof. Thus, macro expansion can be viewed as a local transformation on the syntax tree of a program, where some specific node is replaced by another [without](#) affecting its children (****descendant??**) or parent nodes.

Examples

****This are here as place-holders; they need to be updated per Matan****

Define a macro that creates a new action with the internal name `simple_frame` to generate a frame with specified field values and call a method named `send()`.

```
define <simple_frame'action> "send simple frame \  
  <dest_addr'num> <source_addr'num> <size'num>" as {  
  var f: frame;  
  gen f keeping {  
    .kind not in [SRAM,DUT];  
    .size == <size'num>;  
    .dest_address == <dest_addr'num>;  
    .source_address == <source_addr'num>;  
  };  
  start f.send();  
};
```

The following is a definition of an action with the internal name `swap_var`. The match string contains two parsing element items, `<var1'exp>` and `<var2'exp>`, so the `<1>` in the third line corresponds to `<var1'exp>`, the first parsing element in the match string. The notation `<2>` could likewise be used for `<var2'exp>`. Thus, the third line could be written as `<1> = <2|z>`.

```
define <swap_var'action> "swap <var1'exp>[ <var2'exp>]" as {  
  var tmp<?> := <var1'exp>;  
  <1> = <var2'exp|z>;  
  <var2'exp|z> = tmp<?>;  
};  
extend sys {  
  run() is also {  
    var a:= 5;  
    var b:= 9;  
    var z:= 13;  
    swap a b;           // a becomes 9, b becomes 5  
    print a, b, z;  
    swap a;             // a becomes 13, z becomes 9  
    print a, b, z;  
  };  
};
```