Description of the language MINI

M. Seutter

September 11, 2004

1 Introduction

MINI is a true sublanguage of ELAN, which means that MINI programs can be compiled by existing ELAN compilers such as `elancc`. As data objects only variables and constants of type INT, BOOL or TEXT can be used. Only one control construct exists, namely the WHILE loop allowing repetitive execution of actions. Input and output may be done via the primitive procedures `get` and `put`. These procedures may only be called with one parameter. Simple formulas and assignments can be used to calculate values and to assign these to variables.

The syntax of the language is designed in such a way that language extensions can be made with little pain. An extension of the language MINI is the language MAXI. This language is not a sublanguage of ELAN, although it is possible to write MAXI programs in such a way that they may be compiled correctly by an existing ELAN compiler. One of the goals of this compiler construction course is to write a compiler for MAXI.

The syntax of MINI will be described by a context free grammar, augmented with a number of metarules to improve readability of the grammar. For some of the grammar rules simple examples are given. The context conditions and the semantics for the syntactical constructs will be described verbally if necessary.

2 Abbreviations

Syntax

a)
```
NOTION option:
    NOTION;
    .
```

b) `NOTION sequence:`
```
    NOTION sequence, NOTION;
    NOTION.
```

c) `NOTION list:`
```
    NOTION list, comma token, NOTION;
    NOTION.
```

d) `NOTION pack:`
```
    open token, NOTION, close token.
```

e) `NOTION token:`
```
    comments option, NOTION symbol.
```

3 Program

Syntax

a) `program:`
```
    refinement name, colon token, paragraph, point token.
```

b) `prelude:`
```
    prelude paragraph.
```

Example

```
fac:
   INT VAR i;
   put ("give number:\n");
   get (i);
   INT CONST i plus 1 :: i + 1;
   INT VAR fac :: 1;
   INT VAR counter1 :: 1;
   INT VAR counter2;
   INT VAR sum;
   WHILE counter1 < i plus 1
   REP
      sum := 0;
      counter2 := 0;
      WHILE counter2 < counter 1
      REP
         sum := sum + fac;
```

```
        counter2 := counter2 + 1
      ENDREP;
      counter1 := counter1 + 1
   ENDREP;
   put ("fac (");
   put (text (i));
   put (") = ");
   put (text (fac));
   put ("\n").
```

Context conditions

A program has type VOID (possibly after coercions). A number of procedures and operators are assumed to be predeclared in the standard prelude:

```
INT OP + (INT CONST a, INT CONST b)
INT OP - (INT CONST a, INT CONST b)
INT OP * (INT CONST a, INT CONST b)
INT OP DIV (INT CONST a, INT CONST b)
INT OP MOD (INT CONST a, INT CONST b)
INT OP + (INT CONST a)
INT OP - (INT CONST a)

BOOL OP =  (INT CONST a, INT CONST b)
BOOL OP <> (INT CONST a, INT CONST b)
BOOL OP <  (INT CONST a, INT CONST b)
BOOL OP <= (INT CONST a, INT CONST b)
BOOL OP >  (INT CONST a, INT CONST b)
BOOL OP >= (INT CONST a, INT CONST b)

BOOL OP =   (BOOL CONST a, BOOL CONST b)
BOOL OP <>  (BOOL CONST a, BOOL CONST b)
BOOL OP AND (BOOL CONST a, BOOL CONST b)
BOOL OP OR  (BOOL CONST a, BOOL CONST b)
BOOL OP NOT (BOOL CONST a)

TEXT PROC text (INT CONST a)
TEXT PROC text (BOOL CONST a)

PROC get (TEXT VAR a)
PROC get (INT VAR a)
PROC put (TEXT CONST a).
```

Before parsing a MINI program, the MINI compiler reads the standard prelude from the file `mini-std.prl`.

4 Paragraphs and units

Syntax

a)
```
paragraph:
    proper paragraph;
    .
```

b)
```
proper paragraph:
    proper paragraph, semicolon token, unit;
    unit.
```

c)
```
unit:
    object declaration;
    expression.
```

d)
```
paragraph without declarations:
    proper paragraph without declaration;
    .
```

e)
```
proper paragraph without declarations:
    proper paragraph without declaration,
       semicolon token, expression;
    expression.
```

f)
```
prelude paragraph:
    closed declaration, semicolon symbol,
       prelude paragraph;
    .
```

Context conditions

a,b) A MINI program contains one paragraph. In this paragraph only identifiers of type INT, BOOL or TEXT may be declared. No identifier may be defined more than once. An empty paragraph has type VOID. A nonempty paragraph has the same type and access as the last unit in the paragraph. All other units in the paragraph have type VOID.

c) A unit has the same type and access as the object declaration or expression.

d,e) An empty paragraph without declarations has type VOID. A nonempty paragraph without declarations has the same type and access as the last expression. All other expressions in the paragraph without declarations have type VOID.

Semantics

a,b,c) The declarations and expressions of a paragraph will be executed sequentially from left to right. The result of a paragraph is the result of the last unit.

d,e) The expressions of a paragraph without declarations will be executed sequentially. The result of a paragraph without declarations is the result of the last expression.

5 Control constructs

5.1 The repetition

Syntax

a) `repetition:`
```
    while clause.
```

b) `while clause:`
```
    while part, loop body.
```

c) `while part:`
```
    while token, condition.
```

d) `loop body:`
```
    rep token, paragraph without declarations,
      endrep token.
```

e) `condition:`
```
    expression.
```

Examples

```
WHILE TRUE REP ENDREP
WHILE i > 0 REP get (i); put (text (i)) ENDREP
```

Context conditions

a) A repetition always has type VOID.

d) The paragraph without declarations has type VOID.

e) The condition has type BOOL and access CONST.

Semantics

b) In a while clause, the while part will be evaluated first. If the enquiry clause yields the value TRUE, the loop body will be evaluated. The while part will then be reevaluated and so on. When the enquiry clause yields the value FALSE, the evaluation of the while clause is finished.

6 Declarations and declarers

6.1 Object declarations

Syntax

a) `object declaration:`
```
    constant declaration;
    variable declaration.
```

b) `constant declaration:`
```
    type declarer, const token option, object name,
      initialization.
```

c) `variable declaration:`
```
    type declarer, var token, object name,
      initialization option.
```

d) `initialization:`
```
    initial token, expression.
```

Context conditions

a) An object declaration has type VOID. Multiple object declarations with the same name are not allowed nor is it allowed to declare an object with the same name as a procedure (declared in the standard prelude).

b) A constant declaration fixes the type, name and value of the declared constant. A constant may only be applied after its declaration.

c) A variable declaration fixes the type, name and possibly initial value of the declared variable. A variable may only be applied after its declaration.

6.2 Procedure and operator declarations

Syntax

a) `closed declaration:`
```
    procedure declaration;
    operator declaration.
```

b) `procedure declaration:`
```
    procedure head, routine body, procedure tail.
```

c) `procedure head:`
```
    result option, proc token, procedure name,
       formal parameter pack, colon token.
```

d) `procedure tail:`
```
    endproc token, procedure name option.
```

e) `operator declaration:`
```
    operator head, routine body, operator tail.
```

f) `operator head:`
```
    result, op token, operator name,
       formal parameters pack, colon token.
```

g) `operator tail:`
```
    endop token, operator name option.
```

h) `routine body:`
```
    internal token, text denotation;
    external token, text denotation.
```

i) `result:`
```
    type declarer.
```

j) `formal parameters:`
```
    formal parameter, comma token, formal parameter;
    formal parameter.
```

k) `formal parameter:`
```
    type declarer, access declarer option, parameter name.
```

Context conditions

a) A closed declaration has type VOID. It is not allowed to declare multiple procedures or operators having the same name, number of arguments and same argument types.

b) A procedure declaration fixes the result type, name, argument type, and body of the declared procedure.

d) If present, the procedure name in the procedure tail must match the procedure name in the procedure head.

e) An operator declaration fixes the result type, name, number of arguments, argument types and body of the declared operator.

g) If present, the operator name in the operator tail must match the operator name in the operator head.

h) Internal routine bodies are directly translated by the MINI compiler, as indicated by their internal name. External routine bodies are mapped onto routines in the MINI runtime system, identified by their external name.

k) When no access declarer is present, CONST access is assumed.

Examples

b)
```
TEXT PROC text (INT CONST a):
   EXTERNAL "int_to_text"
ENDPROC;
```

e)
```
INT OP + (INT CONST a, INT CONST b):
   INTERNAL "int plus int"
ENDOP;
```

6.3 Type declarers

Syntax

a)
```
type declarer:
   simple type declarer.
```

b)
```
simple type declarer:
   int token;
   bool token;
   text token.
```

6.4 Access declarers

Syntax

a) `access declarer:`
```
    const token;
    var token.
```

7 Expressions

Syntax

a) `expression:`
```
    assignment;
    formula.
```

Context conditions

a) The expression has the same type and access as the assignment or formula.

7.1 Assignments

Syntax

a) `assignment:`
```
    destination, becomes token, source.
```

b) `destination:`
```
    formula.
```

c) `source:`
```
    formula.
```

Context conditions

a) The assignment has the type VOID. The destination and source must have the same type.

b) The destination has access VAR.

c) The source has access CONST.

Semantics

a) The left and right hand side of the assignment will be evaluated. Afterwards the value calculated for the right hand side will be assigned to the variable of the left hand side.

7.2 Formulas

Syntax

a) `formula:`
```
    monadic formula, dyadic operator, monadic formula;
    monadic formula.
```

b) `monadic formula:`
```
    monadic operator, primary;
    primary.
```

c) `dyadic operator:`
```
    capital name;
    or token;
    and token;
    equal token;
    not equal token;
    less than token;
    less equal token;
    greater than token;
    greater equal token;
    plus token;
    minus token;
    asterix token;
    div token;
    mod token.
```

d) `monadic operator:`
```
    capital name;
    plus token;
    minus token;
    not token.
```

Context conditions

a) The types of the operands of the dyadic operator are determined. Using these types and the name of the dyadic operator, the corresponding operator definition is identified among the predefined operators. The type of the formula is the same as the result type of the identified operator; the access is CONST. The operands of the dyadic operator must have the same type and access as the corresponding formal parameter of the identified operator.

   If the formula is a single monadic formula, the type and access of the formula is the same as that of the monadic formula.

b) The type of the operand of the monadic operator is determined. Using this type and the name of the monadic operator, the corresponding operator definition is identified among the predefined operators. The type of the monadic formula is the same as the result type of the identified operator; the access is CONST. The operand of the monadic operator must have the same type and access as the corresponding formal parameter of the identified operator.

   If the monadic formula is a single primary, the type and access of the monadic formula is the same as that of the primary.

Semantics

a,b) The corresponding operator will be called after the evaluation of the operands. The result of the application of operators should be as expected by the conventions of analysis.

7.3 Primaries and calls

Syntax

a) `primary:`
```
    repetition;
    call;
    denoter;
    identifier application.
```

b) `call:`
```
    procedure name, actual parameter pack.
```

c) `actual parameter:`
```
    expression.
```

d) `identifier application:`
```
    object name.
```

Context conditions

a) The primary has the same type and access as the repetition, call, denoter or identifier application.

b) The type of the actual parameter of the call is determined. Using this type and the procedure name, the corresponding procedure definition is identified among the predefined procedures. The type of the call is the same as the result type of the identified procedure; the access is CONST. The actual parameter of the call must have the same type and access as the corresponding formal parameter of the identified procedure.

d) Only identifiers that were previously declared may be applied. The type and access of an identifier application is the type and access of the object whose declaration defined the identifier. However if the access demanded by the context is CONST and the identified object is a variable, the access of the identifier application is coerced to CONST.

Semantics

b) The corresponding procedure will be called after the evaluation of the argument.

d) An identifier application yields the identified object.

8 Names

Syntax

a) `refinement name:`
```
    small name.
```

b) `object name:`
```
    small name.
```

c) `parameter name:`
```
    small name.
```

d) `procedure name:`
```
    small name.
```

e) `operator name:`
```
    capital name;
    special name.
```

f) `capital name:`
```
    comments option, [A-Z][A-Z0-9]*,
    { A capital name may not be one of the reserved
      words, named in the section 10 }.
```

g) `small name:`
```
    small name, comments option, [a-z0-9]+;
    comments option, [a-z][a-z0-9]*.
```

h) `special name:`
```
    or token;
    and token;
    equal token;
    not equal token;
    less token;
    less equal token;
    greater token;
    greater equal token;
    plus token;
    minus token;
    asterix token;
    div token;
    mod token;
    not token.
```

9 Denoters

Syntax

a) `denoter:`
```
    comments option, denotation.
```

b) `denotation:`
```
    integer denotation;
    boolean denotation;
    text denotation.
```

c) `integer denotation:`
```
    digit sequence.
```

d) `boolean denotation:`
```
    true symbol;
    false symbol.
```

e) `text denotation:`
```
    quote symbol, text item sequence option, quote symbol.
```

f) `digit:`
```
    [0-9].
```

g) `text item:`
```
    ([^\\\n\"]|\\.).
```

Context conditions

a) The type of the denoter is the same as the type of the denotation. The access of a denoter is CONST.

b) The type of a denotation is INT if it is an integer denotation, BOOL if it is a boolean denotation and TEXT if it is a text denotation.

10 Symbols

The following table describes all symbols used in the MINI language. Layout inside the symbols is not allowed.

int symbol	`INT`
bool symbol	`BOOL`
text symbol	`TEXT`
const symbol	`CONST`
var symbol	`VAR`
while symbol	`WHILE`
rep symbol	`REP`
endrep symbol	`ENDREP`
colon symbol	`:`
initial symbol	`::`
becomes symbol	`:=`
semicolon symbol	`;`
point symbol	`.`
comma symbol	`,`

open symbol	`(`
close symbol	`)`
plus symbol	`+`
minus symbol	`-`
asterix symbol	`*`
div symbol	`DIV`
mod symbol	`MOD`
equal symbol	`=`
not equal symbol	`<>`
less than symbol	`<`
less equal symbol	`<=`
greater than symbol	`>`
greater equal symbol	`>=`
and symbol	`AND`
or symbol	`OR`
not symbol	`NOT`
true symbol	`TRUE`
false symbol	`FALSE`
quote symbol	`"`
comment open symbol	`{`
comment close symbol	`}`

The following table describes all additional symbols allowed in the standard prelude:

proc symbol	`PROC`
endproc symbol	`ENDPROC`
op symbol	`OP`
endop symbol	`ENDOP`
internal symbol	`INTERNAL`
external symbol	`EXTERNAL`

11 Comments and layout

Syntax

a) `comments:`
```
    comments option, comment, layout;
    layout.
```

b) `comment:`
```
    comment open symbol, comment chars,
      comment close symbol.
```

c) `comment chars:`
```
    [^{}]*.
```

d) `layout:`
```
    [ \t\n\r]*.
```

Example

```
{ This is a comment }
```

Semantics

Comments are meaningless for the execution of the program. They only serve as documentation for human readers of programs.

12 Coercions

There are two coercions in MINI, namely:

1) consting: when demanded by the context the access of a variable application is forced to CONST. This yields the value of the variable.

2) voiding: may be applied to any expression. The value of the expression is forgotten. The result has the type VOID.