Formal Languages and Compilers
Proff. Breveglieri, Crespi Reghizzi, Morzenti
Written exam[1]: laboratory question
06/02/2009

SURNAME: .....
NAME: ..... Student ID: .....
Course: ○ Laurea Specialistica ○ V. O. ○ Laurea Triennale ○ Other:.....
Instructor: ○ Prof. Breveglieri ○ Prof. Crespi ○ Prof Morzenti

The laboratory question must be answered taking into account the implementation of the `Acse` compiler given with the exam text.

Modify the specification of the lexical analyzer (`flex` input) and the syntactic analyzer (`bison` input) and any other source file required to extend the `Lance` language with the ability to *handle simple macros* resembling `#define` construct for the C preprocessor :

```
define ANSWER 42;
define QUESTION 9;
int x;
read( x );
x = ANSWER * x;
write( x );
```

The first line of the sample code snippet defines the macro `ANSWER` as the integer value `42`. The expected behaviour of this program is to print the number provided in input multiplied by 42.

The solution needs to comply to the following specifications:

- The macros are parameter-free and each macro can only bind to a single *integer value*
- An arbitrary number of macros may be defined
- Macros may be employed *everywhere* an integer constant can be used in the original `Acse` language
- Macros cannot be modified by assignments

An **optimal solution** shouldn't generate any additional assembly code for a `Lance` program using macros with respect to one which isn't.

In case a double definition of the same macro is detected, implement a sensible strategy (either ignore or overwrite the former definition).

[1]Time 45'. Textbooks and notes can be used.
Pencil writing is allowed. Write your name on any additional sheet.

You may specify any further (sensible) assumption useful to complete the given specification list.

You may use the functions in `collections.h` if you are in need of a set of ready-made helpers for dealing with lists. If you prefer, you may also use these functions in order to handle lists of `DATA` typed elements. In the latter case, define the structure `DATA`.

```
void initList( t_list *list );
void addFirst( t_list *list, DATA *element );
void addLast( t_list *list, DATA *element );
DATA *getFirstElement( t_list *list );
DATA *getLastElement( t_list *list );
DATA *getElementAt( t_list *list, unsigned int position );
```

1. Define the tokens (and the related declarations in Acse.lex e Acse.y). (3 points)

   In **Acse.lex** add:

   ```
   "define"        { return DEFINE; }
   ```

   In **Acse.y**:

   ```
   %token DEFINE
   ```

2. Define the syntactic rules or the modifications required to the existing ones. (8 points)

   macro_def : DEFINE IDENTIFIER NUMBER SEMICOLON ;

   macro_defs : *macro_defs macro_def*
   | ;

   Modify the existing rule:

   program : *macro_defs var_declarations statements* ;

   Add new alternatives to existing rules:

   declaration : ...
   | IDENTIFIER ASSIGN IDENTIFIER
   | IDENTIFIER LSQUARE IDENTIFIER RSQUARE
   ;

   exp : ...
   | NOT_OP IDENTIFIER
   ;

3. Define the semantic actions needed to implement macros. (13 points for a working solution, 19 for the optimal one)

In case of duplicate definition, we report an error.

In **Acse.y**:

```
%{
   ...
   t_list macros;
%}
...
typedef struct {
  char *id;
  int val;
} DATA;

...

int main() {
   initList( &macros );
   ...
}
```

Let us assume that we have the function with prototype:

DATA *getMacro(const char *id);

This function returns the structure of the macro identified by `id`, or `NULL` if the macro does not exist.

Semantic actions:

```
macro_def : DEFINE IDENTIFIER NUMBER SEMICOLON
    {
        DATA *p;
        p = getMacro( $2 );
        if (p) {
            /* Report a duplicate-definition error */
        }
        p = malloc( sizeof(DATA) );
        addList( &macros, p );
        p->id = $2;
        p->val = $3;
    }
  ;
```

```
declaration : IDENTIFIER ASSIGN IDENTIFIER
    {
        DATA *p = getMacro( $3 );
        if (! p) {
            /* Report an error */
        }
        $$ = alloc_declaration( $1, 0, 0, p->val );
    }
 |  IDENTIFIER LSQUARE IDENTIFIER RSQUARE
    {
        DATA *p = getMacro( $3 );
        if (! p) {
            /* Report an error */
        }
        $$ = alloc_declaration( $1, 1, p->val, 0 );
    }
  ;

exp : IDENTIFIER
    {
        DATA *p = getMacro( $1 );
        if (! p) {
            /* Code of the old action */
        } else {
            $$ = create_expression( p->val, IMMEDIATE );
        }
    }
 |  NOT_OP IDENTIFIER
    {
        DATA *p = getMacro( $1 );
        if (! p) {
            /* Report an error */
        }
        $$ = create_expression( p->val == 0, IMMEDIATE );
    }
  ;
```

4. Bonus: Modify the existing solution in order to allow recursive definition of macros; for instance, the following program

```
define ULTIMATE_ANSWER  42;
define ANSWER  ULTIMATE_ANSWER;
int x;
x = ANSWER;
write( x );
```

prints "42". (5 points)

Add an alternative:

```
macro_def : ...
 |  DEFINE IDENTIFIER IDENTIFIER SEMICOLON
    {
        DATA *p1, p2;
        p1 = getMacro( $2 );
        if (p1) {
            /* Report a duplicate-definition error */
        }
        p1 = malloc( sizeof(DATA) );
        p2 = getMacro( $3 );
        if (! p2) {
            /* Report a undefined-macro error */
        }
        addList( &macros, p1 );
        p1->id = $2;
        p1->val = p2->val;
    }
  ;
```

A solution like the one given above is little verbose, but it is worth the maximum grade. Here some more details are given to help students.

The solution forces macro definitions to appear before any code. A solution that considers a macro definition as any other instruction has problems. For example,

statement : ...
  | *macro_def*
;

allows the production of code whose semantics is not clear, like this:

```
if (x == 3)
   define MACRO 45;
```

The compiler always considers the macro definition (it does not know the value of `x`, so this is the only option), but this is contrary to the semantics of the *if* statement.

All actions are at end rule; this and the use of a global macro list `macros` avoid the need of introducing new semantic attributes.

The function `getMacro()` can be written as:

```
DATA *getMacro( const char *id ) {
    unsigned int i = 0;
    DATA *p;

    do {
        p = getElementAt( &macros, i );
        if (0 != strcmp( id, p->id ))
            return p;
        ++i;
    } while (p != NULL);
    return NULL;
}
```

where we assume that `getElementAt()` returns `NULL` when the given index is past the end of the list.

The given solution does not check if there is clash between a macro and a variable definition. This check could have been introduced at the beginning of the actions of the non-terminal *declaration* in this way:

```
DATA *p1;
p1 = getMacro( $1 );
if (p1) {
    /* Report a duplicate-definition error */
}
...
```

It is possible to write a more general solution, which complies also with the specifications of the bonus question. This solution defines *macro_def* in this way:

macro_def : DEFINE IDENTIFIER *exp* ;

This solution allows to define the value of a macro as a constant expression. The semantic action is:

```
macro_def : DEFINE IDENTIFIER exp
    {
        DATA *p;
        if ($3.expression_type != IMMEDIATE) {
            /* Report an error: expression is not constant */
        }
        p = getMacro( $2 );
        if (p) {
            /* Report a duplicate-definition error */
        }
        p = malloc( sizeof(DATA) );
        addList( &macros, p );
        p->id = $2;
        p->val = $3.value;
    }
  ;
```