N.V.PHILIPS Natuurkundig Laboratorium

NATLAB
NR : TPC
DD : 81/01/22
PAR : EO

TWIN PASCAL COMPILER

N.V.PHILIPS Natuurkundig Laboratorium

NATLAB
NR : TPC
DD : 81/01/22
PAR : EO

TWIN PASCAL COMPILER

1. Introduction

In this paper the Twin Pascal compiler is described. The paper is intended to facilitate compiler maintenance, to describe the generated object code and to illustrate the correctness of the compiler.

The compilation of a source module occurs in two passes. In the first pass the names, types and values are collected; in the second pass the object code is generated.

The compiler produces code for BSM in relocatable form. The Twin Link Editor is to be used to form load modules that can be executed on the Twin System.

2. General

2.1. Source input

The source input is supported by a subroutine. This routine gets the next symbol from the current source line (by skipping comments, blanks and line markers) and it determines the type of the symbol.

When the current source line is exhausted, this line is printed first during pass II (if a listing is wanted). Next the following line is retrieved form the source file.

The recognized symbol types are:

- identifier;
- integer number;
- real number;
- character string;
- special symbol (conform to Twin Pascal);
- end of file.

The syntax rules of Pascal are used to determine the symbols.

2.2. Recovery from syntax errors

A syntax error appears when the compiler encounters an unexpected symbol. Recovery from such errors can be done (after signalling the error) by:

- insertion of the missing symbol;
- replacing the faulty symbol by the expected one;
- skipping source text until the expected symbol is found.

The Twin Pascal compiler uses all these methods, but skipping of source text occurs conditionally. Skipping stops upon the recognition of a so called "stop symbol". The used stop symbols depend on the state of the compilation; it is possible to add symbols to the list of stop symbols and to remove symbols from this list.

2.3. Object code

The produced object code consists of records of different types. The type is identified by the first byte of the record. The type identifiers with their meaning are:

H'80': ESD-record, not SD;
H'81': ESD-record, SD;
H'00': TXT-record;
H'01': RLD-record;
H'FF': END-record.

The record formats are:

```
ESD: byte   0:      record id, H'80' or H'81'
            1-6:    symbol, left aligned and padded with blanks
            7:      type:  H'80': SD - section definition
                           H'00': ER - external reference
                           H'01': CD - absolute constant definition
                           H'02': LD - entry definition
                           H'03': LD - label definition
                           H'04': LD - relocatable constant definition
                           H'05': CD - variable definition
            8-9:    value: SD: origin of section
                           ER: sequence number
                           LD: address in section
                           CD: value
            10-11:  SD only, number of bytes in section.

TXT: byte   0:      record id, H'00'
            1:      number of text bytes
            2-3:    address of text in section
            4 up:   up to 28 bytes of object text.

RLD: byte   0:      record id, H'01'
            1:      number of data bytes
            2 up:   data items with format:
                    bit    0-2:  type: 00X - 8 bit byte
                                       11X - 15 bit address
                                       XX0 - local reference
                                       XX1 - external reference
                           3-7:  offset in following string of text
                           8-15: present only with external references,
                                 sequence number of ESD-record.

 ND: byte   0:      record id, H'FF';
            1-2:    bit    0:    type: 0 - entry
                                       1 - no entry
                           1-15: entry point if bit 0 is 0.
```

2.4. Symbollist

The symbollist contains two classes of data: identifier definitions and type definitions. The identifier definitions are grouped into sublists. There exists a sublist for the program parameters, for the fields in each record and for the identifiers defined in each block. The format of an identifier definition is:

```
  bytes 0-1:  address of next identifier definition (high-order byte
is 0 with
        2:    type                                                last
definition)
        3-4:  address of type definition
        5-6:  address
        7 up: identifier, terminated by CR
```

The general format of a type definition is:

```
bytes 0:     number of bytes in type definition
      1:     type
      2 up:  type dependent data
```

Subroutines exist to:

- open a new sublist;
- close the current sublist (and to proceed with the enveloping sublist);
- add an identifier definition to the current sublist;
- add a type definition to the symbollist;
- locate an identifier in the current sublist and enveloping sublists;
- locate an identifier in the current sublist only;
- locate an identifier in a given sublist;
- step to the next identifier definition (if any) in a sublist;
- locate a type definition from various sources.

2.5. Types

Each programming element (identifier, expression etc) is characterised by its elementary type; an identifier is qualified by a general type too. The general types are:

```
CON:    constant;
LBL:    label;
VAR:    variable;
PVAR:   variable parameter;
FIELD:  record field;
TYPE:   type;
SPROC:  standard procedure;
PROC:   user procedure;
SFUN:   standard function;
FUN:    user function.
```

The elementary types are:

```
BOOL:   boolean;
CHAR:   character;
SCAL:   enumerated scalar;
INT:    integer number;
REAL:   real number;
SET:    set;
PTR:    pointer or label;
FILE:   file;
ARRAY:  array;
REC:    record;
TID:    descriptor reference;
UNSP:   unspecified.
```

Also subranges of BOOL, CHAR, SCAL and INT can be indicated.

In order to simplify the type compatibility checks, the elementary type TID is used in many cases. This type is mainly a reference to another item in the symbollist (which may be of the type TID).

Two types are compatible if:

1. Both types refer to the same type definition, or:
2. If one of the following statements is true for the (refered) types:
    1. Both types are:
        - BOOL, inclusive subranges of BOOL, or
        - INT, inclusive subranges of INT, or
        - REAL;
    2. Both types are one of the types CHAR (or a subrange thereof) and ARRAY[n] OF CHAR (string), and the lengthes of both types are the same;
    3. Both types are SCAL, inclusive subranges of SCAL, with the same type id;
    4. One type is INT (or a subrange thereof) and the other type is REAL;
    5. Both types are PTR and a component type is UNSP;
    6. Both types are SET and a base type is UNSP;
    7. Both types are SET with compatible base types.

3. Expressions

3.1. Introduction

An expression is translated conform to the following syntax rules:

```
<expression>          ::= <simple expression> <rel> <simple expression>
                        ! <simple expression>

<simple expression>   ::= <sign> <term> <add> <term>
                        ! <sign> <term>

<term>                ::= <factor> <mult> <factor>
                        ! <factor>

<factor>              ::= ( <expression> )
                        ! NOT <factor>
                        ! <set>
                        ! unsigned constant
                        ! variable
                        ! function

<set>                 ::= [ <set element> ]

<set element>         ::= <expression> .. <expression> , <set element>
                        ! <expression> , <set element>
                        ! nil

<mult>                ::= * ! / ! DIV ! MOD ! AND

<add>                 ::= + ! - ! OR

<sign>                ::= + ! - ! nil

<rel>                 ::= = ! <> ! < ! <= ! > ! => ! IN
```

3.2. Relation operations

A relation operation can be executed for a pair of operands that are simple expressions. The operands must be of compatible types. The type of the first operand must also satisfy the following requirements:

operation	allowed types
<, >	BOOL, CHAR, SCAL, INT, REAL
<=, >=	BOOL, CHAR, SCAL, INT, REAL, SET
=, <>	BOOL, CHAR, SCAL, INT, REAL, SET, PTR

With the operation IN the allowed types for the first operand are

BOOL, CHAR, SCAL and INT. With this operation the type of the second operand must be SET; the base type of this set must be compatible with the type of the first operand.

The object code contains one of the instructions TE, TLT, TGT, TNE, TLE, TGE, TIN to execute the operation. The type of the result is always BOOL.

3.3. Addition and multiplication operations

An addition operation can be executed for one or two operands which are terms of an expression; a multiplication operation can be executed for two operands that are factors of an expression. When two operands are involved, these operands must be of compatible types. The type of the first, or only, operand has also to satisfy the following requirements:

unary operations	allowed types
unsigned	all types
+, -	INT or subranges thereof, REAL
NOT	BOOL or subranges thereof

binary operations	allowed types
+, -, *	INT or subranges thereof, REAL, SET
/	INT or subranges thereof, REAL
DIV, MOD	INT or subranges thereof (both operands)
AND, OR	BOOL or subranges thereof (both operands)

The object code contains one of the instructions CPL, NOT, ADD, SUB, MULT, RDIV, WDIV, MOD, AND or OR to execute the operation.

If one of the operand types is REAL, the type of the result is REAL. The type of the result is always REAL with the operation "/". In other cases the type of the first operand is used as the type of the result.

The operand descriptors are replaced by the descriptor of the result type.

3.4. Set

The translation of a set begins with the generation of instructions to construct an empty set. The base type of this set is UNSP.

When the set is not empty, the expressions are translated. After translation of the first expression the base descriptor is replaced by the descriptor of the expression and this type is checked. The allowed (refered) types are:

BOOL,
CHAR (only with length 1),

SCAL,
INT,

or subranges of these types.

The types of following expressions in the set must be valid base types and they must be compatible with the derived base type.

If an expression of type CHAR is encoutered, the instructions:

```
LODI    A' '
SUB
```

are generated in order to point to the right bit in the set.

When a set element refers to a single bit, the instruction SETB (set bit) is stored in the object code; for set elements refering to a bit string, the instruction SETBS (set bit string) is inserted.

3.5. Functions

Two types of functions are distinguished, "standard functions" and "user functions". The standard functions are the TWIN Pascal functions, a user function is described in a function declaration.

3.5.1. User functions

When a reference to an user function is found, first the instruction CDSA (create DSA) is generated. This instruction indicates also the number of bytes in the function result.

Next the parameter list is processed. All parameters, conform to the heading of the function declaration, must be present. When the formal parameter is a value parameter, an expression is translated, if the formal parameter is a variable parameter, a variable is processed.

The type of the actual parameter must be compatible with the type of the formal parameter, but an INT formal parameter and a REAL actual parameter are not allowed.

After processing of the parameter list the instruction JPS (jump to subroutine), refering to the entry of the function, is added to the object code.

The type of the function result is TID. The descriptor refers to the function descriptor.

3.5.2. Standard functions

An operand is optional with the standard functions EOLN and EOF. If an operand is present it must be a variable of the type FILE. The default

operand is the standard input file. The result of these functions is of the type BOOL.

The standard function RAND must not have an operand. This function generates a REAL number.

All other standard functions require an operand, which is an expression. The allowed operand types and the result type depend on the function, as follows:

function	operand types	result type
CHR	INT	CHAR
ARCTAN, COS, EXP, LN, TAN, SIN, SQRT	INT, REAL	REAL
ABS, SQR	INT, REAL	operand type
TRUNC, ROUND	REAL	INT
ODD	INT	BOOL
ORD	BOOL, CHAR, SCAL, INT	INT
SUCC, PRED	BOOL, CHAR, SCAL, INT	operand type

For all standard functions an operation code consisting of the code ESC and the function index is created.

With the functions SUCC and PRED the operation code is preceded by instructions to load the limit. With the function SUCC the limit is the highest possible value of the operand; with the function PRED the limit is the lowest possible value.

3.6. Unsigend constants

An unsigned constant can be:

- the special symbol NIL;
- a constant identifier;
- a self defining constant (number or character string).

3.6.1. Special symbol NIL

When the special symbol NIL is a factor in an expression, instructions to load a word with the value 0 are generated and a pointer descriptor with UNSP component type is produced.

3.6.2. Constant identifiers and self defining constants

Constant identifiers and self defining constants in expressions are translated in the same way as constants in the constant definition part. The instruction LODI, followed by the value of the constant, is added to the object code.

3.7. Variables

A variable starts with either a field identifier (in a WITH statement) or a variable identifier. First is tested on a field identifier.

The general type corresponding to a variable identifier can be VAR or PVAR. With variable parameters the address of the variable points to the location where the actual address is stored. To load this address, the following instructions are generated:

```
LODA    <address>          load pointer to variable address
LODD                       load address of variable
```

With field identifiers the following instructions are produced:

```
LODA    <rec.addr ptr>     load pointer to record address
LODD                       load address of record
LODI    <field offset>     load offset of field in record
ADD                        add address and offset
```

For other variables only the first instruction (LODA) is formed.

With all types a reference to the identifier type definition is used as the descriptor of the variable.

When the type of the (partial) translated variable is ARRAY, index expressions may follow; if the type is REC then field designators are allowed; when the type is PTR or FILE a reference to the component type can be made.

To refer to a component (allowed with the types PTR and FILE), the instruction LODD is produced, because the variable points to the address of the component. The variable descriptor is replaced by a reference to the component type.

When a field identifier is present (allowed with the type REC) the following instructions are generated:

```
LODI    <field offset>     load offset of field in record
ADD                        add record address and offset
```

The descriptor of the variable is replaced by a reference to the type definition of the designated field.

When index expressions are present (permitted with the type ARRAY) instructions are produced to address the designated array component. This occurs as follows:

1. An expression is translated, the expression type must be compatible with the index type but the combination of REAL expression and INT index is not allowed;

2. If the index type is a subrange and the lower bound of the subrange is not 0, the following instructions are formed:

```
        LODI    <lower bound>     subtract lower bound
        SUB
```

3. If the index type is CHAR, but not a subrange, the next instructions are generated:

```
        LODI    A' '              subtract lower bound
        SUB
```

4. The instruction LODI is added to the object code;

5. The instructions MULT and ADD are generated;

6. Finally, the array descriptor is replaced by (a reference to) the type definition of the array component.

If a next index expression is present, this expression is processed.

4. Programs

Twin Pascal programs are compiled in the form of modules. A module is either a main program or a procedure module. A main program consists of a program header and a complete block; a procedure module contains only a block, but not the statement part of the block.

The generated object file contains the translation of the compiled block. With the main program this block is followed by the next BSM instructions:

```
        STOP                                stop Pascal program
ENTRY   BA      4                           jump to begin of BSM
        ISP     H'01,24'                      reserve room for standard
files
        CFCB    input file descriptor
        CFCB    output file descriptor
        CDSA,0                              create DSA for Pascal program
        JPS,0   Pascal entry                jump to Pascal program
```

The label ENTRY identifies the entry point of the load module.

4.1. Label definitions

The declared labels are only stored in the symbollist. Labels declared at level 0 are also added to the External Symbol Dictionary. Labels in the main program are external definitions, while labels in procedure modules are external references.

The bytes 2-6 of a label definition represent:

```
  bytes 1:    LBL, the type PTR is added upon assignment of the
ddress
        3-4:  0
        5-6:  address of statement (SSA-address)
```

4.2. Constant definitions

Constant defined in the constant definition part are stored in the symbollist. All such constants, except those defined at level 0 of procedure modules, are also written in the object file.

The format of a constant type definition is:

```
  bytes 0:    count
        1:    type (BOOL, CHAR, SCAL or INT)
        2-3:  type id with SCAL
        4 up: constant
```

Constants defined at level 0 are also represented in the External ymbol Dictionary. Constants in a main program are external definitions, constants in procedure modules are external references.

4.3. Type definitions

4.3.1. General

A type defines the set of values which variables of the type may assume and the operations on these variables. Type definitons are only stored in the symbollist.

Pointer types may refer to a type that is defined lateron in the type definition part. In this case an UNSP type definition is generated. As a result of this, multiple specified type definitions may appear in a type definition part. If a previously defined type identifier, of type UNSP, is encountered, the type definition is updated and a reference to this type definition is made.

4.3.2. Array

An array definiton consists of a number of index types followed by a component type. The index types are simple types which must define a limited number of discrete values. The component type may be any type. The format of an array descriptor is:

```
bytes 0:    count
      1:    ARRAY
      2-3:  number of bytes in array
      4 up: index type followed by component type
```

The component type may be an array type. This occurs when more than one index type is specified.

4.3.3. Pointer

A pointer type defines a reference to a variable of a given type (the component type). A pointer type definition may refer to a component type that is specified lateron in the type definition part. In this case a component type definition of the type UNSP is generated. This component type definition will be completed when the actual component type definition is compiled. The format of a pointer descriptor is:

```
bytes 0:   4
      1:   PTR
      2-3: address of component type definition in symbollist
```

4.3.4. File

A file type defines a file together with the type of the elements in the file. A file type cannot be a part of any other type definition.

A file is represented by a File Control Block (FCB). A FCB occupies 16

bytes. The first two bytes in a FCB contain the address of the file buffer.

The elements in a file may be of any type. A special class of files are textfiles, defined by:

```
TEXT = FILE OF CHAR;
```

With textfiles room is to be reserved for a complete line of text. A line may occupy 129 bytes (including the line marker). The format of a file descriptor is:

```
bytes 0:    count
      1:    FILE
      2-3:  length of FCB (16) + length of component type
      4 up: component type                    (130 for textfiles)
```

The maximum length of a file component is 255.

4.3.5. Set

A set type defines the range of values which is the power set of its base type. The base type is a simple type that must define a limited number of discrete values. The limit is 64.

When the base type of a set is a subrange, the lower bound of the subrange must be the lowest possible value of the unlimited type, e.g. 0 for subranges of INT and "space" for subranges of CHAR.

The format of a set descriptor is:

```
bytes 0:    count
      1:    SET
      2 up: base type
```

4.3.6. Record

A record type is a structured type consisting of a fixed number of components, called "fields", possibly of different types.

Fields in a record are accessible only through a reference to the record. Therefore, the field identifiers are collected in a sublist.

The format of a record descriptor is:

```
bytes 0:    6
      1:    REC
      2-3:  number of bytes in record
      4-5:  address of sublist with field identifiers
```

4.3.7. Non-standard scalar

An non-standard scalar defines a set of values by enumeration of the identifiers which denote these values. The definition of an non-standard scalar comprises (besides the type definition) also a number of constant definitions, associated to the value identifiers. The lowest value is 0. The format of the corresponding constant identifier definitions is:

```
bytes 0-1:  address of next definition
      2:    CON
      3-4:  address of type definition
      5-6:  value
      7 up: identifier, terminated by CR
```

The format of a non-standard scalar type definition is:

```
bytes 0:    6
      1:    SCAL
      2-3:  address of type definition
      4-5:  value of last constant, i.e. number of elements - 1
```

4.3.8. Subrange

A subrange defines a set of values which is a subrange of another simple type, called "host type". Possible host types are BOOL, CHAR, SCAL and INT. The format of a subrange descriptor is:

```
bytes 0:    count
      1:    SUBR + host type
      2-3:  address of host type definition
      4 up: lower limit, followed by upper limit
```

The limits of subranges of CHAR occupy one byte, the limis of subranges of other host types occupy two bytes.

4.4. Variable declarations

A variable declaration is an ordinary identifier definition (see 2.4).

Variables defined at level 0 are also represented in the External Symbol Dictionary. Variables in the main program are external definitions, variables in procedure modules are external references.

4.5. Procedure and function declarations

A procedure declaration associates an identifier with a part of a program that can be activated by a procedure statement. A function declaration associates an identifier with a part of a program that computes a value of a simple type or a pointer type. Procedure declarations and function declarations are similar, except that a function declaration must specify the result type.

With each procedure and function declaration the static level is increased, a new sublist in the symbollist is opened and the state of the DSA location counter is set to 0.

Procedures and functions may have parameters. Only value parameters (except FILE's) and variable parameters are allowed in Twin Pascal.

The parameters specified in the procedure (function) heading are stored as the first variables in the new sublist. The descriptors of value parameters and variable parameters are similar, but the type of variable parameters is PVAR.

The result type of a function has to be a type identifier which refers to one of the following types (or subranges thereof):

- BOOL;
- CHAR;
- SCAL;
- INT;
- REAL;
- PTR.

Procedure headers and function headers may be followed by one of the directives FORWARD or EXTERN.

The directive FORWARD indicates that the procedure (function) block is given lateron. EXTERN specifies that the block will be compiled separately. External references are allowed only at level 0.

Procedure identifiers and function identifiers declared at level 0 appear also in the External Symbol Dictionary. These are external definitions, except when they are declared as EXTERN.

The format of a procedure (function) definition is:

```
bytes 0:      12
      1:      PROC or FUN
      2-3:    0 with PROC, address of type definition with FUN
      4:      number of bytes in function result
      5:      bit 0:   forward indicator
                  1-7: number of parameters
      6-7:    anchor of sublist
      8-9:    number of bytes in DSA
      10-11:  DSA location count
```

After the compilation of the procedure (function) block the instruction RET (return) is written in the object file.

4.6. Statement part

A statement part shall not occur at level 0 of procedure modules.

As first instruction of a statement part the instruction ISP

(increment stack pointer) is written in the object file. The operand of this instruction designates the number of bytes used by the variables (except parameters) declared in the block.

The instruction ISP is followed by a number of CFCB instructions (one for each file variable declared in the block). The format of a CFCB instruction is:

```
bytes 0:    operation index CFCB
      1-2:  address of FCB (DSA address)
      3:    bits 0:    textfile indicator
                 1-7:  parameter number (external files)
      4:    number of bytes in file buffer (130 for textfiles)
      5 up: file name, terminated by CR
```

e description of a statement part consists of a sequence of statements. Two statements are separated by the symbol ';'. The statement part is terminated by the symbol 'END'.

A statement may be preceded by a label (specified by an integer number followed by a colon). The label must be declared in the corresponding label definition part. When a label is encountered, the label definition is completed.

If with the compilation of a source text a fourth parameter is specified, the instruction TRACE is generated as the first instruction of each statement.

A statement begins with a special symbol or an identifier. The special symbol must designate a structured statement or a GOTO statement. The identifier must be a variable identifier, a function identifier or a procedure identifier. A variable identifier and a function identifier imply an assignment statement; the procedure identifier specifies a procedure statement.

4.6.1. Assignment statement

Assignment of a value is possible to a variable and to the result of a function. The type of the left hand part of an assignment statement is the result of the translation of the variable or it is a reference to the function result type. The type shall not be (a reference to) a file.

The type of the right hand part is generated as a part of the translation of the expression. Both types shall be compatible, but assignment of a real value to an item of the type INT is forbidden.

The translation of the expression is followed by the instruction STF with the static level of the function definition (by assignments to function identifiers) or by the instruction STD (with assignments to variables).

4.6.2. Procedure statements

Two types of procedures are distinguished: "standard procedures" and "user procedures". The standard procedures are the standard Pascal procedures, a user procedure is described in a procedure declaration.

4.6.2.1. User procedures

The translation of a procedure statement that refers to a user procedure begins with the production of the instruction CDSA (create DSA), with 0 for the number of bytes in the function result. Then the parameters (if any) are translated (see user functions). Finally the instruction JPS (jump to subroutine), refering to the entry of the procedure, is formed.

4.6.2.2. Standard procedures

The executed functions depend on the standard procedure identifier, as follows:

4.6.2.2.1. DISPOSE, PACK and UNPACK

Source text is skipped until one of the symbols ";", "END", "ELSE" or "UNTIL".

4.6.2.2.2. NEW

The procedure NEW requires a single parameter which must be a PTR variable. This variable is translated. Next the following instructions are produced:

```
LODI    <component length>      load length of component
ESC                             create new variable
NEW
```

4.6.2.2.3. PAGE

A parameter is optional with the procedure PAGE. When a parameter is present, the procedure statement is translated in the same way as the procedure statements GET, PUT, RESET and REWRITE. If no parameter is given, then the translation occurs conform to the procedure statement WRITELN.

4.6.2.2.4. READLN and WRITELN

A parameter is optional. The first, or only, parameter may be a FILE variable. The default file variable is INPUT (READLN), respectively OUTPUT (WRITELN).

When no parameter is given or the only parameter is a FILE variable, instructions are generated to load the address of the (default) file variable (FCB) and to invoke the corresponding standard procedure.

When there is more than one parameter or the only parameter is not a file variable, the translation proceeds in the same way as the procedure statements READ and WRITE.

4.6.2.2.5. READ and WRITE

The first parameter of these procedures may be a file variable. The default file variable is INPUT (READ), respectively OUTPUT (WRITE).

These procedures require, besides the optional file reference, at least one data parameter. The data parameters are variables (READ), respectively expressions (WRITE).

When the designated file is not a textfile, the type of the variable (expression) must be compatible with the file component type, but combinations of INT and REAL are not allowed.

With the procedure READ and a textfile, the following variable types (or subranges thereof) are accepted:

CHAR;
INT;
REAL.

With the procedure WRITE and a textfile, the next expression types (or subranges thereof) are allowed:

BOOL;
CHAR;
INT;
REAL;
ARRAY[n] OF CHAR (string).

With the procedure WRITE an expression may be followed by field width parameters; 2 field width parameters are permitted with the expression type REAL, 1 field width parameter is allowed with other expression types. Field width parameters are expressions which must be of the type INT. When no field width parameters are given, default indicators are produced.

After the translation of each data parameter (with possible field width parameters), instructions are produced to load the address of the file variable and to invoke the corresponding standard procedure (either READ or WRITE). With the procedure READ these instructions are followed by the instruction STD (store datum).

When the last parameter has been processed and the designated procedure is READLN or WRITELN, next instructions are generated to load the address of the file variable and to invoke the corresponding

standard procedure (either READLN or WRITELN).

4.6.2.2.6. GET, PUT, RESET and REWRITE

These procedure statements require a single parameter that must be of the type FILE. The translation of the variable is followed by instructions to invoke the designated standard procedure.

4.6.3. GOTO statement

A GOTO statement must refer to a defined label. A GOTO statement is translated into an unconditional jump to the address assigned to the label.

4.6.4. Structured statements

Structured statements are constructs composed of other statements.

4.6.4.1. IF statement

An IF statement begins with an expression. The type of this expression must be BOOL. With the translation of an IF statement without an ELSE clause the following object code is generated:

```
        <expression>                    evaluate expression
        JPF         label1              jump if false
        <statement>                     THEN statement
label1: . . . . .
```

For an IF statement with an ELSE clause the next object code is produced:

```
        <expression>                    evaluate expression
        JPF         label1              jump if false
        <statement>                     THEN statement
        JP          label2
label1: <statement>                     ELSE statement
label2: . . . . .
```

4.6.4.2. WHILE statement

A WHILE statement begins with an expression. The type of this expression must be BOOL. The following object code is produced with the translation of a WHILE statement:

```
label1: <expression>                    evaluate expression
        JPF         label2              jump if false
        <statement>
        JP          label1
```

label2: . . . . .

4.6.4.3. REPEAT statement

A REPEAT statement begins with a sequence of statements. The last statement is followed by an expression. The type of this expression must be BOOL. With the translation of a REPEAT statement the following object code is generated:

```
label1: <statement>                     sequence of statements
             !
             !
        <statement>
        <expression>                    evaluate expression
        JPF          label1             jump if false
```

4.6.4.4. WITH statement

A WITH statement begins with a string of variables which must be of the type REC. The address of the sublist with field identifiers of each record is saved, so the fields of these records are direct accessible. Also additional (unnamed) variables are created to save the addresses of the record variables.

The object code contains the following data for a WITH statement:

```
        <variable>                      determine record addresses
             !
        <variable>
        <statement>                     WITH statement
```

After processing of the statement the addresses of the sublists with field identifiers are destroyed.

4.6.4.5. CASE statement

A CASE statement begins with an expression. The allowed expression types are BOOL, CHAR, SCAL, INT and subranges of these types.

Next follows a string of case list elements. A case list element is a collection of case labels followed by a statement, the special symbol ELSE followed by a statement, or empty. The ELSE form must be the last case list element of a CASE statement.

A case label is a constant. The constant type must be compatible with the expression type, but REAL constants are not accepted.

For a case list element, consisting of a collection of case labels and a statement, the following object code is produced:

```
        CSEL         label1 label2      case select
```

```
        <constant>                          case labels
             !
        <constant>
label1: <statement>                         statement
        JP          label3                  jump to next statement
label2: . . . . .
```

For a case list element of the form ELSE <statement> the next object code is generated:

```
label2: DSP         expr length             erase result of expression
        <statement>                         ELSE statement
label3: . . . . .
```

When the ELSE form of the case list element is omitted, the following instructions are formed at the end of a CASE statement:

```
label2: DSP         expr length             erase result of expression
label3: . . . . .
```

4.6.4.6. FOR statement

A FOR statement begins with an ordinary assign statement. The allowed types of the left variable are BOOL, CHAR, SCAL, INT and subranges of these types.

Next follows one of the special symbols TO or DOWNTO and an expression denoting the final value of the control variable. The expression type must be compatible with the type of the control variable, but REAL expressions are not allowed. An (unnamed) variable is created to keep the final value.

The expression is followed by a statement.

The following object code is generated with the translation of a FOR statement:

```
        <assign>                                initial assignment
        LODA        DSA loc counter             address of final value
        <expression>                            final value
label1: LODA        control variable            load value of control
variable
        LODD
        LODA        DSA loc counter             load final value
        LODD
        TLE                                     TGE with DOWNTO
        JPF         label2                      jump if final value passed
        <statement>                             execute statement
        LODA        control variable            assign next value to control
        LODA        control variable                                variable
        LODD
        LODI        1
        ADD                                     SUB with DOWNTO
        STD
        JP          label1
label2: . . . . .
```