The ACK Modula-2 Compiler

Ceriel J.H. Jacobs

Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands

1. Introduction

This document describes the implementation-specific features of the ACK Modula-2 compiler. It is not intended to teach Modula-2 programming. For a description of the Modula-2 language, the reader is referred to [1].

The ACK Modula-2 compiler is currently available for use with the VAX, Motorola MC68020, Motorola MC68000, PDP-11, and Intel 8086 code-generators. For the 8086, MC68000, and MC68020, floating point emulation is used. This is made available with the *-fp* option, which must be passed to *ack*[4,5].

2. The language implemented

This section discusses the deviations from the Modula-2 language as described in the "Report on The Programming Language Modula-2", as it appeared in [1], from now on referred to as "the Report". Also, the Report sometimes leaves room for interpretation. The section numbers mentioned are the section numbers of the Report.

2.1. Syntax (section 2)

The syntax recognized is that of the Report, with some extensions to also recognize the syntax of an earlier definition, given in [2]. Only one compilation unit per file is accepted.

2.2. Vocabulary and Representation (section 3)

The input "`10..`" is parsed as two tokens: "`10`" and "`..`".

The empty string `""` has type

```
ARRAY [0 .. 0] OF CHAR
```

and contains one character: `0C`.

When the text of a comment starts with a '`$`', it may be a pragma. Currently, the following pragmas exist:

```
(*$F       (F stands for Foreign) *)
(*$R[+|-] (Runtime checks, on or off, default on) *)
(*$A[+|-] (Array bound checks, on or off, default off) *)
(*$U       (Allow for underscores within identifiers) *)
```

The Foreign pragma is only meaningful in a `DEFINITION MODULE`, and indicates that this `DEFINITION MODULE` describes an interface to a module written in another language (for instance C, Pascal, or EM). Runtime checks that can be disabled are: range checks, `CARDINAL` overflow checks, checks when assigning a `CARDINAL` to an `INTEGER` and vice versa, and checks that `FOR`-loop control-variables are not changed in the body of the loop. Array bound checks can be enabled, because many EM implementations do not implement the array bound checking of the EM array instructions. When enabled, the compiler

generates a check before generating an EM array instruction. Even when underscores are enabled, they still may not start an identifier.

Constants of type `LONGINT` are integers with a suffix letter `D` (for instance `1987D`). Constants of type `LONGREAL` have suffix `D` if a scale factor is missing, or have `D` in place of `E` in the scale factor (f.i. `1.0D`, `0.314D1`). This addition was made, because there was no way to indicate long constants, and also because the addition was made in Wirth's newest Modula-2 compiler.

2.3. Declarations and scope rules (section 4)

Standard identifiers are considered to be predeclared, and valid in all parts of a program. They are called *pervasive*. Unfortunately, the Report does not state how this pervasiveness is accomplished. However, page 87 of [1] states: "Standard identifiers are automatically imported into all modules". Our implementation therefore allows redeclarations of standard identifiers within procedures, but not within modules.

2.4. Constant expressions (section 5)

Each operand of a constant expression must be a constant: a string, a number, a set, an enumeration literal, a qualifier denoting a constant expression, a type transfer with a constant argument, or one of the standard procedures `ABS`, `CAP`, `CHR`, `LONG`, `MAX`, `MIN`, `ODD`, `ORD`, `SIZE`, `SHORT`, `TSIZE`, or `VAL`, with constant argument(s); `TSIZE` and `SIZE` may also have a variable as argument.

Floating point expressions are never evaluated compile time, because the compiler basically functions as a cross-compiler, and thus cannot use the floating point instructions of the machine on which it runs. Also, `MAX(REAL)` and `MIN(REAL)` are not allowed.

2.5. Type declarations (section 6)

2.5.1. Basic types (section 6.1)

The type `CHAR` includes the ASCII character set as a subset. Values range from `0C` to `377C`, not from `0C` to `177C`.

2.5.2. Enumerations (section 6.2)

The maximum number of enumeration literals in any one enumeration type is `MAX(INTEGER)`.

2.5.3. Record types (section 6.5)

The syntax of variant sections in [1] is different from the one in [2]. Our implementation recognizes both, giving a warning for the older one. However, see section 3.

2.5.4. Set types (section 6.6)

The only limitation imposed by the compiler is that the base type of the set must be a subrange type, an enumeration type, `CHAR`, or `BOOLEAN`. So, the lower bound may be negative. However, if a negative lower bound is used, the compiler gives a warning of the *restricted* class (see the manual page of the compiler).

The standard type `BITSET` is defined as

```
TYPE BITSET = SET OF [0 .. 8*SIZE(INTEGER)-1];
```

2.6. Expressions (section 8)

2.6.1. Operators (section 8.2)

2.6.1.1. Arithmetic operators (section 8.2.1)

The Report does not specify the priority of the unary operators + or −: It does not specify whether

```
- 1 + 1
```

means

```
- (1 + 1)
```

or

```
(-1) + 1
```

I have seen some compilers that implement the first alternative, and others that implement the second. Our compiler implements the second, which is suggested by the fact that their priority is not specified, which might indicate that it is the same as that of their binary counterparts. And then the rule about left to right decides for the second. On the other hand one might argue that, since the grammar only allows for one unary operator in a simple expression, it must apply to the whole simple expression, not just the first term.

2.7. Statements (section 9)

2.7.1. Assignments (section 9.1)

The Report does not define the evaluation order in an assignment. Our compiler certainly chooses an evaluation order, but it is explicitly left undefined. Therefore, programs that depend on it may cease to work later.

The types `INTEGER` and `CARDINAL` are assignment-compatible with `LONGINT`, and `REAL` is assignment-compatible with `LONGREAL`.

2.7.2. Case statements (section 9.5)

The size of the type of the case-expression must be less than or equal to the word-size.

The Report does not specify what happens if the value of the case-expression does not occur as a label of any case, and there is no `ELSE`-part. In our implementation, this results in a runtime error.

2.7.3. For statements (section 9.8)

The Report does not specify the legal types for a control variable. Our implementation allows the basic types (except `REAL`), enumeration types, and subranges. A runtime warning is generated when the value of the control variable is changed by the statement sequence that forms the body of the loop, unless runtime checking is disabled.

2.7.4. Return and exit statements (section 9.11)

The Report does not specify which result-types are legal. Our implementation allows any result type.

2.8. Procedure declarations (section 10)

Function procedures must exit through a RETURN statement, or a runtime error occurs.

2.8.1. Standard procedures (section 10.2)

Our implementation supports `NEW` and `DISPOSE` for backwards compatibility, but issues warnings for their use. However, see section 3.

Also, some new standard procedures were added, similar to the new standard procedures in Wirth's newest compiler:

- `LONG` converts an argument of type `INTEGER` or `REAL` to the types `LONGINT` or `LONGREAL`.
- `SHORT` performs the inverse transformation, without range checks.
- `FLOATD` is analogous to `FLOAT`, but yields a result of type `LONGREAL`.
- `TRUNCD` is analogous to `TRUNC`, but yields a result of type `LONGINT`.

2.9. System-dependent facilities (section 12)

The type `BYTE` is added to the `SYSTEM` module. It occupies a storage unit of 8 bits. `ARRAY OF BYTE` has a similar effect to `ARRAY OF WORD`, but is safer. In some obscure cases the `ARRAY OF WORD` mechanism does not quite work properly.

The procedure `IOTRANSFER` is not implemented.

3. Backwards compatibility

Besides recognizing the language as described in [1], the compiler recognizes most of the language described in [2], for backwards compatibility. It warns the user for old-fashioned constructions (constructions that [1] does not allow). If the *-Rm2-3* option (see [6]) is passed to *ack*, this backwards compatibility feature is disabled. Also, it may not be present on some smaller machines, like the PDP-11.

4. Compile time errors

The compile time error messages are intended to be self-explanatory, and not listed here. The compiler also sometimes issues warnings, recognizable by a warning-classification between parentheses. Currently, there are 3 classifications:

(old-fashioned use)
: These warnings are given on constructions that are not allowed by [1], but are allowed by [2].

(strict)
: These warnings are given on constructions that are supported by the ACK Modula-2 compiler, but might not be supported by others. Examples: functions returning structured types, SET types of subranges with negative lower bound.

(warning)
: The other warnings, such as warnings about variables that are never assigned, never used, etc.

5. Runtime errors

The ACK Modula-2 compiler produces code for an EM machine as defined in [3]. Therefore, it depends on the implementation of the EM machine for detection some of the runtime errors that could occur.

The *Traps* module enables the user to install his own runtime error handler. The default one just displays what happened and exits. Basically, a trap handler is just a procedure that takes an INTEGER as parameter. The INTEGER is the trap number. This INTEGER can be one of the EM trap numbers, listed in [3], or one of the numbers listed in the *Traps* definition module.

The following runtime errors may occur:

array bound error
: The detection of this error depends on the EM implementation.

range bound error
: Range bound errors are always detected, unless runtime checks are disabled.

set bound error
: The detection of this error depends on the EM implementation. The current implementations detect this error.

integer overflow
: The detection of this error depends on the EM implementation.

cardinal overflow
: This error is detected, unless runtime checks are disabled.

cardinal underflow
: This error is detected, unless runtime checks are disabled.

real overflow
: The detection of this error depends on the EM implementation.

real underflow
: The detection of this error depends on the EM implementation.

divide by 0
: The detection of this error depends on the EM implementation.

divide by 0.0
: The detection of this error depends on the EM implementation.

undefined integer
: The detection of this error depends on the EM implementation.

undefined real
: The detection of this error depends on the EM implementation.

conversion error
: This error occurs when assigning a negative value of type INTEGER to a variable of type CARDINAL, or when assigning a value of CARDINAL that is > MAX(INTEGER), to a variable of type INTEGER. It is detected, unless runtime checking is disabled.

stack overflow
: The detection of this error depends on the EM implementation.

heap overflow
: The detection of this error depends on the EM implementation. Might happen when ALLOCATE fails.

case error
: This error occurs when non of the cases in a CASE statement are selected, and the CASE statement has no ELSE part. The detection of this error depends on the EM implementation. All current EM implementations detect this error.

stack size of process too large
: This is most likely to happen if the reserved space for a coroutine stack is too small. In this case, increase the size of the area given to `NEWPROCESS`. It can also happen if the stack needed for the main process is too large and there are coroutines. In this case, the only fix is to reduce the stack size needed by the main process, f.i. by avoiding local arrays.

too many nested traps + handlers
: This error can only occur when the user has installed his own trap handler. It means that during execution of the trap handler another trap has occurred, and that several times. In some cases, this is an error because of overflow of some internal tables.

no RETURN from function procedure
: This error occurs when a function procedure does not return properly ("falls" through).

illegal instruction
: This error might occur when floating point operations are used on an implementation that does not have floating point.

In addition, some of the library modules may give error messages. The **Traps**-module has a suitable mechanism for this.

6. Calling the compiler

See [4,5,6] for a detailed explanation.

The compiler itself has no version checking mechanism. A special linker would be needed to do that. Therefore, a makefile generator is included [7].

7. The procedure call interface

Parameters are pushed on the stack in reversed order, so that the EM AB (argument base) register indicates the first parameter. For VAR parameters, its address is passed, for value parameters its value. The only exception to this rule is with conformant arrays. For conformant arrays, the address is passed, and an array descriptor is passed. The descriptor is an EM array descriptor. It consists of three fields: the lower

bound (always 0), upper bound - lower bound, and the size of the elements. The descriptor is pushed first. If the parameter is a value parameter, the called routine must make sure that its value is never changed, for instance by making its own copy of the array. The Modula-2 compiler does exactly this.

When the size of the return value of a function procedure is larger than the maximum of `SIZE(LONGREAL)` and twice the pointer-size, the caller reserves this space on the stack, above the parameters. Callee then stores its result there, and returns no other value.

8. References

[1] Niklaus Wirth, *Programming in Modula-2, third, corrected edition,* Springer-Verlag, Berlin (1985)

[2] Niklaus Wirth, *Programming in Modula-2,* Stringer-Verlag, Berlin (1983)

[3] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, E.G.Keizer, *Description of a machine architecture for use with block structured languages,* Informatica rapport IR-81, Vrije Universiteit, Amsterdam

[4] UNIX manual *ack*(1)

[5] UNIX manual *modula-2*(1)

[6] UNIX manual *em_m2*(6)

[7] UNIX manual *m2mm*(1)