Implementation of an imperative programming language with backtracking

Doctoraalscriptie van
Vincent Partington
`vincentp@xs4all.nl`

Supervisor:
Prof.Dr. K.R. Apt
`apt@cwi.nl`

July 1997

Faculty of Mathematics, Computer Science, Physics and Astronomy
University of Amsterdam
Plantage Muidergracht 24
1018 VT Amsterdam

Preface

The paradigm of imperative programming is well known; the first programming language most people learn is an imperative one, and imperative programming languages are often used for everyday programming. Unfortunately, this causes many people to miss out on the advantages of logic programming, especially were it concerns search algorithms.

In this report we explore the possibility of combining the paradigms of imperative and logic programming, by implementing a compiler for an imperative language with the extensions proposed in [AS97]. These extensions bring some of the advantages of logic programming to imperative programming by introducing the notions of *failure* and *backtracking*, which have proven so successful in logic programming languages like Prolog.

Part I gives a quick overview of the features of this new programming language called Alma-0, which was based upon Modula-2 [Wir85], and describes the abstract machine used to implement Alma-0 (the Alma Abstract Machine, abbreviated to AAA), which combines the features of abstract machines for imperative languages with those of abstract machines for logical languages.

Part II describes the implementation of the Alma-0 compiler in detail, from lexical analysis to code generation and the run-time system. It not only describes the current implementation, but commentary sections describe the problems encountered during the implementation and the motivations behind certain solutions. These sections have been divided from the main text by large captions, and can be recognized by the little "c" after the section number.

I would like to thank Krzysztof Apt for his support, and for the enlightening conversations I've had with him on many a rainy morning, and, if one of the mornings was not rainy, it should have been, just for the effect. My fellow students should be thanked for their interesting remarks and discussions, whether they concerned my project or not. Finally, I would especially like to thank my parents and Michelle for keeping me going, and for telling me to get on with it.

Contents

Part I

Design

Chapter 1

The Alma-0 programming language

In this chapter we assume the reader is familiar with Modula-2, or a similar imperative programming language, and describe the peculiar features of Alma-0; the extensions, the input/output procedures, the unsupported Modula-2 features, and the small changes when compared to Modula-2. See appendix A for an overview of the syntax of Alma-0 and appendix B for some examples of Alma-0 programs.

1.1 Extensions

The most interesting aspect of the Alma-0 programming language are the extensions it provides. The extensions are based on the extensions proposed in [AS97], and a short description of each is given in the following sections:

1.1.1 The BES extension (boolean expression as statement)

Alma-0 allows the programmer to use a boolean expression as a statement:

- If the expression evaluates to `TRUE`, execution continues after the statement.
- If the expression evaluates to `FALSE`, we say the statement *fails*, or a *failure* has occurred.
- If no failure occurs during execution of a sequence of statements, we say the sequence of statements *succeeds*, otherwise it fails.

An example is the test `waste < TotalValue - CurrentBest` on line 28 of `knapsack.a0` (see section B.1), which tests whether the current solution is better than the current best.

1.1.2 The SBE extension (statements as boolean expression)

Alma-0 allows the programmer to use a sequence of statements as a boolean expression:

- If the sequence of statements succeeds, the expression evaluates to `TRUE`
- If the sequence of statements fails, the expression evaluates to `FALSE`

An example is the call to the procedure `Squares` on line 49 of `squares.a0` (see section B.4), which causes a solution to be printed, only when one was found.

1.1.3 The ORELSE extension

The `ORELSE` statement

```
EITHER s ORELSE t ORELSE u END;
```

starts by executing `s`. If execution fails, either in that branch or beyond the end of the `ORELSE` statement, all assignments done since the beginning of the `ORELSE` statement are undone, and execution continues with `t`. If execution fails again, all assignments are undone again, and execution continues with `u`. If execution fails this time, no special action is performed by the `ORELSE` statement.

The mechanism by which execution can continue, is that of the *choice point*; when the `ORELSE` statement is executed a choice point is created. When a statement fails, *backtracking* is performed, i.e. all assignment done since the choice point are undone, and execution continues at the choice point.

An example is the `ORELSE` statement on lines 21–29 of `knapsack.a0` (see section B.1), which first adds the object to the knapsack, and, should that fail, removes the object.

1.1.4 The SOME extension

The `SOME` statement can be seen as an iterated `ORELSE` statement. The statement

```
SOME i := a TO b DO s END;
```

is equivalent to:

- `FALSE`, when `a > b`.
- `s`, when `a = b`.
- `EITHER i := a; s`
  `ORELSE SOME i := a+1 TO b DO s END;`
  when `a < b`.

An example is the `SOME` statement on lines 28–33 of `squares.a0` (see section B.4), which causes all squared to be tried on the current position.

1.1.5 The COMMIT extension

The `COMMIT` statement prevents superfluous backtracking from happening. The statement

```
COMMIT s END;
```

first executes `s`, and then deletes all choice points created during the execution of `s`.

1.1.6 The FORALL extension

The `FORALL` statement allows the programmer to explore all the possibilities of a sequence of statements. The statement

```
FORALL s DO t END;
```

executes `s`, and, as long as there are choice points left in `s`, backtracking is performed to the last choice point. The `FORALL` statement succeeds when there are no choice points left in `s`, even when `s` failed. After each successful completion of `s`, `t` is executed. Backtracking is not performed over the statements in `t`, which makes it possible to record the current state without losing it when backtracking occurs in `s`.

An example is the `FORALL` statement on lines 25–27 of `queens.a0` (see section B.3), which causes all solutions to the queens problem to be found.

1.1.7 The EQ extension

With every variable of basic type, Alma-0 associates a flag, which signifies whether the variable is *initialized* or *uninitialized*. Initially, a variable is uninitialized, and only after a value has been assigned to it, does it become initialized. Actually, this flag is associated with every lvalue of a basic type, e.g. every element of an array, provided the element is of a basic type.

A related concept is that of *knownness*; if all variables in a expression are initialized, the expression has a *known* value, and otherwise it has an *unknown* value.

When an uninitialized variable is used in an expression, a run-time error is generated. The only exception to this rule are the equality operator =, and the *call-by-mixed-form* parameter passing mechanism described in the next section.

In Alma-0, the semantics of the operator = are different from those of the other relational operators. The behavior of the comparison `s = t` depends upon `s` and `t`:

- If `s` and `t` are expressions with a known value, a regular comparison is performed.
- If `s` is an uninitialized variable, and `t` is an expression with a known value, the value of `t` is assigned to `s`.
- If `t` is an uninitialized variable, and `s` is an expression with a known value, the value of `s` is assigned to `t`.
- All remaining cases generate a run-time error.

An example is the test `e = a[i]` on line 8 of `present.a0` (see section B.2), which tests whether `e` is an element of the array `a`, or, if e is uninitialized, sets `e` to every element of the array `a`, upon backtracking.

1.1.8 The MIX extension

To allow for the use of a procedure parameter as an input parameter *and* as an output parameter, the *call-by-mixed-form* parameter passing mechanism has been provided.

From the perspective of the procedure being called (the *callee*), call-by-mixed-form is identical to call-by-variable.

From the perspective of the code calling the procedure (the *caller*), call-by-mixed-form can behave as call-by-variable or as call-by-value, depending upon the value being passed:

- When an lvalue is passed, call-by-mixed-form is identical to call-by-variable.
- When a value is passed, which is not an lvalue, call-by-mixed-form is similar to call-by-value; the value is stored in a temporary, invisible variable, and this temporary variable is passed to the procedure. When the parameter is changed by the callee, only the temporary variable is changed.

An example is the `MIX` declaration on line 5 of `present.a0` (see section B.2), which allows `e` to be an lvalue or a regular value.

1.1.9 The KNOWN extension

In order to determine, whether an lvalue of a basic type has a known value, the built in function `KNOWN` can be used. The value of `KNOWN(x)` depends on `x`:

- If `x` is an initialized lvalue of a basic type, `KNOWN(x)` equals `TRUE`.
- If `x` is an uninitialized lvalue of a basic type, `KNOWN(x)` equals `FALSE`.
- In all other cases, `KNOWN(x)` yields a compile-time error.

Note that this definition does not correspond precisely to the definition of knownness given above; the knownness of a complicated expression like `x+y` cannot be tested. Instead the variables in the expression need to be tested separately, e.g. `KNOWN(x); KNOWN(y)`. See section 7.2.10 for the motivations behind this discrepancy.

An example is the `KNOWN` statements on lines 12–13 of `squares.a0` (see section B.4), which tests whether the current square has been placed yet.

1.2 Input and output

Alma-0 has three built in procedures which provide support for input and output operations:

- `READ` reads a number of values from the standard input stream and assigns them to the variables given as parameters. The number of values read equals the number of parameters, which should be more than zero.
- `WRITE` writes the values of its parameters to the standard output stream. There should be at least one parameter.
- `WRITELN` is similar to `WRITE`, but `WRITELN` appends a newline character to the output, and `WRITELN` can also be called with zero parameters.

Examples are the `WRITE` and `WRITELN` statements on lines 64–69 of `knapsack.a0` (see section B.1), and the `READ` statement on line 22 of `present.a0` (see section B.2).

1.3 Missing features

Features that are present in Modula-2 but that are missing in Alma-0 include:

- The `CARDINAL` type, sets, variant parts in records, open array parameters, procedure types, and pointer types.
- The `LOOP`, `EXIT`, `CASE`, and `WITH` statements.
- Nested procedures.
- Modules, and therefore the `EXPORT` and `IMPORT` declarations.

1.4 Small differences

A number of small differences can be noticed when comparing Alma-0 to Modula-2:

- Floating point constants are required to have a least one digit after the point; use `1.0` instead of `1.`.
- The operator `REM` is provided as an alternative to the `MOD` operator. It functions identically.
- The inequality operator `#` can also be written as `<>`.
- The syntax of the `REPEAT` statement is different to allow for the use of a sequence of statement as the boolean expression; use

  ```
  REPEAT s UNTIL t END;
  ```

  instead of

  ```
  REPEAT s UNTIL t;
  ```

- Alma-0 allows procedures to return values of structured types (e.g. records and arrays) as well as values of basic types.

Chapter 2

The Alma Abstract Architecture

The Alma Abstract Architecture (AAA) is the virtual architecture used during the intermediate code generation phase of the Alma-0 compiler. Although the current implementation of the AAA entails translating the AAA instructions into C statements, the design of the AAA is such, that it should be possible to translate them into machine code. To this end, the following design criteria were formulated:

- It should be easy for the compiler to select the correct AAA instructions (simple intermediate code generation).
- It should be easy to convert AAA instructions into C statement (simple C code generation).
- The AAA should resemble actual CPU architectures to make it plausible that AAA instructions could be translated into machine code (simple machine code generation).

As the Alma-0 language itself, the AAA aims to combine the best of both worlds; elements were taken from virtual machines used to compile imperative languages (in particular the architecture described in [Wir96, p.55]), and from a virtual machine used to compile a logical language (the WAM [AK91]).

Still, the AAA most resembles the virtual machines used in the compilation of imperative languages. The additions made to provide for the extensions of the Alma-0 language are:

- The failure handling instructions `ONFAIL`, `FAIL`.
- The log control instructions `CREATELOG`, `REPLAYLOG` and `REWINDLOG`.
- The automatic recording of old values in assignment instructions `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `MOVE`, and `CLEAR`.

2.1 Data

2.1.1 Types

The AAA supports the following four data types:

`byte` an 8-bit byte, corresponding to the Alma-0 type `CHAR` and the C type `char`.

`word` the natural word size for the host computer, usually 4 bytes. This type should be large enough to contain all values of the Alma-0 types `INTEGER`, and large enough to hold a pointer.

float a floating point value, corresponding to the Alma-0 type `REAL` and the C type `double`.

string a string value, used for the `WRITE` instruction.

The `string` type is only used by the `immstring` addressing mode and the `WRITE` instruction. It has been provided to allow for efficient compilation of statements like

```
WRITELN('Hello world');
```

Instead of converting the string `'Hello world'` into an array of characters and generating a loop that prints all the characters, a call to the C function `printf()` can be generated, with the entire string as its parameter.

2.1.2 Registers

The AAA has eight registers of type `word`:

`Z` always contains the value zero.

`S1` a scratch register for *very* temporary values.

`S2` another scratch register.

`LP` the log pointer register. It contains an opaque value used by the run-time system to handle log administration; one should only write values to it that have been read from it before, or let the `CREATELOG`, `REPLAYLOG`, and `REWINDLOG` instructions handle this register.

`BP` the failure frame pointer register contains a pointer to the last failure frame allocated on the stack[1]. Failure frames are created by the `ORELSE`, `SOME`, and `FORALL` statements, as well as when a sequence of statements is used as a boolean expression. They hold the saved values of a number of registers (depending upon the statement that created the frame), and the address of the failure handler (see section 2.3).

`EP` the environment frame pointer register contains a pointer to the last procedure call stack frame (comparable to the frame pointer found in actual CPU architectures). Environment frames are created when a procedure is called, and hold the actual parameters, the saved values of a number of registers, the return address, and the local variables.

`SP` the stack pointer points to the top of the stack and is always lower[2] than both `BP` and `EP`.

`PC` the program counter is a virtual register manipulated by the flow control instructions.

2.2 Instructions

An instruction is composed of one *opcode* and three *operands*. The opcode specifies the operation to perform, the operands specify the data on which to perform the operation.

[1]The abbreviation for this register is "BP" instead of "FP" for two reasons; the abbreviation "FP" is usually reserved for the frame pointer, which is more like the AAA's EP register, and "B" is the name of the register in the WAM, that provides a similar function.

[2]The AAA does not stray from the tradition of letting stacks grow backward

2.2.1 Operands

An operand consists of an *addressing mode*, a value, and, sometimes, the name of a register. The addressing mode specifies how the value and the register name should be interpreted:

- The `immbyte` addressing mode indicates that the operand represents a constant `byte` value, e.g. `5` or `'a'`.
- The `immword` addressing mode indicates that the operand represents a constant `word` value, e.g. `70000`.
- The `immfloat` addressing mode indicates that the operand represents a constant `float` value, e.g. `3.1415927`.
- The `immstring` addressing mode indicates that the operand represents a constant `string` value, e.g. `'hello world'`.
- The `regnval` addressing mode indicates that the operand represents a register, possibly incremented by a `word` value, e.g. `S1`, `BP + 70` or `24`.
- The `indbyte` addressing mode indicates that the operand represents a `byte` value at the memory location indicated by the value of a register, possibly incremented by a `word` value, e.g. `byte[EP+10]`.
- The `indword` addressing mode indicates that the operand represents a `word` value at the memory location indicated by the value of a register, possibly incremented by a `word` value.
- The `indfloat` addressing mode indicates that the operand represents a `float` value at the memory location indicated by the value of a register, possibly incremented by a `word` value.
- There is no `indstring` addressing mode.

2.2.2 Opcodes

The AAA has a total of 27 different operations for arithmetic, flow control, log control and I/O (see table 2.1). There are some points to note:

- Of every instruction, which can assign a new value to a memory location, i.e. `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `MOVE` and `CLEAR`, there is a *recording version*, a version that first records the current value of the target memory location in the log (see section 2.3).
- The branch instructions in the AAA combine the separate comparison and branch instructions normally found in actual CPU architectures. A status register is therefore not needed, and more efficient C code can be generated.
- The `LAB` instruction inserts a label in the C code generated by the Alma-0 compiler. It is the C compiler's responsibility to calculate the correct address for the branch instruction, thereby relieving the compiler of that task.
- The I/O instructions were added to the architecture to support Alma-0's built in `READ`, `WRITE` and `WRITELN` procedures. The I/O instructions are directly translated into calls to C `stdio` functions.

Instruction	Pseudo-code
Arithmetic	
`ADD a, b, c`	`a := b + c;`
`SUB a, b, c`	`a := b - c;`
`MUL a, b, c`	`a := b * c;`
`DIV a, b, c`	`a := b / c;`
`MOD a, b, c`	`a := b % c;`
`CHK a, b, c`	`IF a < b OR a > c THEN` *generate a run-time error* `;`
`MOVE dst, src, len`	copy `len` bytes from `mem[src]` to `mem[dst]`
`CLEAR dst, _, len`	set `len` bytes at `mem[dst]` to zero
Comparison & flow control	
`BEQ a, b, l`	`IF a = b THEN GOTO l;`
`BNE a, b, l`	`IF a <> b THEN GOTO l;`
`BLT a, b, l`	`IF a < b THEN GOTO l;`
`BGE a, b, l`	`IF a >= b THEN GOTO l;`
`BGT a, b, l`	`IF a > b THEN GOTO l;`
`BLE a, b, l`	`IF a >= b THEN GOTO l;`
`BRA _, _, l`	`GOTO l;`
`JSR _, _, l`	`mem[EP] := PC; GOTO l;`
`RTS _, _, _`	`GOTO mem[EP];`
`ONFAIL _, _, l`	`mem[BP] := l;`
`FAIL _, _, _`	`GOTO mem[BP];`
`LAB _, _, l`	insert label `l` in output code
`NOP _, _, _`	do nothing
Log control	
`CREATELOG _, _, _`	create a new log
`REPLAYLOG _, _, _`	replay current log and discard it
`REWINDLOG _, oldlp, _`	discard logs without replaying, until `LP` equals `oldlp`
I/O	
`READ a, _, _`	read a value from the standard input stream and store it in `a`
`WRITE _, val, _`	write value `val` to the standard output stream
`WRITELN _, _, _`	write a newline character to the standard output stream

- `_` indicates an operand that is not used and may therefore be of any addressing mode.
- `b` and `c` can be of any addressing mode, except `immstring`.
- `a` can be of any addressing mode, except `immstring`, and must be an lvalue, which excludes the addressing mode `regnval`, when the offset does not equal `0` or the register is `Z`.
- `l` is a label and must be of addressing mode `immword`.
- `src` and `dst` are addresses and must be of addressing mode `immword`.
- `len` is a length parameter and must be of addressing mode `immword`.
- `oldlp` is an opaque `LP` value and must of addressing mode `immword` or `indword`.
- `val` can be of any addressing mode.

Table 2.1: AAA instruction set

2.3 Backtracking

2.3.1 Choice points, failure handling, and log creation

An important difference, one notices, when comparing the AAA to the WAM, is the division of the *choice point* notion into the separate notions of *failure handling* and *log creation* which, when taken together, can be used to implement a choice point.

When a *failure handler* is installed by the `ONFAIL` instruction, the location at which execution should continue in case of a failure, is saved. When a failure is subsequently generated by the `FAIL` instruction, execution continues at the previously saved location. Compare the failure handling notion to the exception handling mechanism in languages such as C++ [ES90] and Java [GJS96]. It is used in the Alma-0 compiler to implement the **BES** and **SBE** extensions.

When a *log* is created by the `CREATELOG` instruction, from that point on, every value that is about to be changed is recorded in the log, but only when the recording version of an instruction is used. When the log is played back by the `REPLAYLOG` instruction, the recorded values are restored. The log can be compared to the trail described in [AK91], and is used in the Alma-0 compiler to implement the **OR**, **SOME**, **FORALL** and **COMMIT** extensions.

A choice point that offers a choice between two execution branches, can be created by creating a new log, setting a failure handler, and executing the first branch. When a failure occurs, the failure handler will be called, which should replay the log and execute the second branch.

2.3.2 The log

More than one log may have been created at one time, but only one log is the *active log*. When a value is recorded, it is recorded in the active log, if there is one. The log administration system behaves as follows:

- At the beginning of the execution of an Alma-0 program there is no (active) log.
- When a log is created by the `CREATELOG` instruction, then the currently active log is deactivated, and the new log becomes the active log.
- When the recording version of an instruction is executed, the current value of the target is saved in the active log, if any, before the assignment is performed.
- When a log is replayed by the `REPLAYLOG` instruction, the values which have been recorded in the log are restored, and the log previously deactivated is made active again (if there was no previous log, there is no longer an active log). Finally, the log just replayed is discarded.
- When the `REWINDLOG` instruction is executed, the active log is discarded and the previous log is activated, until the log indicated by the second operand is active. The values in the discarded logs are *not* restored.

As we can see, the logs behave mostly like a stack of logs. However, the `FORALL` statement breaks the analogy; when execution of the `DO` part starts, the active log is remembered, and the log, which was active before the `FORALL` statement, is activated. When execution of the `DO` part is finished, the log that was remembered is activated again, and any logs created during execution of the `DO` part are discarded.

Part II

Implementation

Chapter 3

Overview of the implementation

The Alma-0 compiler was written in ANSI C [KR88]. The actual development was done on an Apple Macintosh computer with the Metrowerks CodeWarrior development environment, but the compiler has been built successfully with the gcc C compiler on Digital Unix, FreeBSD, Irix, Linux, and Solaris platforms. The Flex [Pax95] and Bison [DS95] tools were used to generate the scanner and parser.

3.1 Compiler structure

It is useful to divide the compilation process into phases. It makes it easier to think about the compilation process as a whole, it gives us a convenient way to divide the compiler into separate, though interdependent, modules, and it provides us with an order for the chapters of this part.

As described in [ASU86, Chapter 1], we can distinguish the *lexical analysis*, *syntax analysis*, *semantic analysis*, *intermediate code generation*, *code optimization*, and *code generation* phases of the compiler. Apart from the optimization phase, all these phases are present in the Alma-0 compiler and are described in the next sections, as are the various utility functions that are used throughout the compiler.

3.1.1 Lexical analysis

During the lexical analysis phase, the *lexical analyzer* divides the source file (also referred to as the *input stream*) into small chunks of data, called *tokens* or *terminals*. This process is also called *tokenizing*. The *lexical rules* tell the lexical analyzer which characters make up the tokens, e.g. the characters `WHILE` make up a keyword, the characters `3.1415927` make up a floating point number, etc. The lexical analysis phase transforms the input stream from a stream of characters into a stream of tokens, which is passed on to the next phase.

The lexical analyzer in the Alma-0 compiler was implemented using Flex [Pax95], the GNU version of the well known lexical analyzer generation tool Lex [Les75]. The file `a0gram.l` contains the regular expressions that are equivalent to the lexical rules. When Flex is run, it reads the file `a0gram.l`, generates a lexical analyzer that behaves according to the specified regular expressions, and writes it to the C source file `lex.yy.c`.

3.1.2 Syntax analysis

During the syntax analysis phase, the stream of tokens generated by the lexical analyzer, are used by the *parser* to form larger constructs, called *non-terminals*. These constructs are hierarchical, i.e. one construct can contain several others, and the top-level construct is the complete source file, the *compilation unit*. The *syntax* tells the parser how to form these constructs from tokens and other constructs, e.g. an addition is formed by an expression, followed by a + token and another expression.

The parser in the Alma-0 compiler was implemented using Bison, the GNU version of the well known parser generation tool Yacc [Joh78]. The file `a0gram.y` contains the context-free syntax rules that make up the syntax. When Bison is run, it reads the file `a0gram.y`, generates a LALR, bottom-up, parser that behaves according to the specified syntax, and writes it to the C source file `y.tab.c`.

3.1.3 Semantic analysis

During the semantic analysis phase, the constructs found by the parser are examined and their validity is checked. Mostly this entails type analysis, but it is also in this phase that constant folding and variable allocation takes place.

During *type analysis* every variable, value and expression is tagged with its type, and every use of each of these is checked against the typing rules of the language. For example, it is forbidden to assign an `INTEGER` value to a `BOOLEAN` variable, because the types are not *compatible*.

Constant folding is the process whereby expressions, which can be evaluated at compile-time are in fact evaluated. For example, the compiler can see that the expression `30+2*5` will always have the value `40`, and it can substitute that value for the expression. Not only does this make the resulting program run quicker, but there are some places where constant folding is absolutely necessary; the dimensions of an array need to be known at compile-time, although they may be given as complicated expressions.

For every variable, a memory location must be allocated, which contains the variable. The *variable allocation* process must decide whether a variable should be stored in global memory (if it's a global variable) or on the stack (if it's a local variable), and it must take care not to allocate one memory location to more than one variable.

In the Alma-0 compiler, type checking is implemented in the C source files `a0type.c` and `a0type.h`, while constant folding and variable allocation are implemented in the C source file `a0value.c` and `a0value.h`

3.1.4 Intermediate code generation

During the intermediate code generation phase, the constructs found in the syntax analysis phase are translated into sequences of AAA instructions, with help from the information gathered in the semantic analysis phase. The hierarchical structure of the syntax is used here; when an addition is translated, the instructions generated for the subexpressions are used to calculate the operands for the addition, and then an addition instruction is generated to perform the actual addition.

In the Alma-0 compiler, intermediate code generation is handled by the C source files `a0code.c`, `a0code.h`, `a0code_flow.c`, and `a0code_ops.c`.

3.1.5 Code generation

The AAA instructions generated in the previous phase can't be executed directly by a CPU. Therefore, during the code generation phase of the compiler, the AAA instructions are translated into a C program (a.out.c), which can be compiled with any ANSI C compiler.

Most compilers emit machine language instructions in the code generation phase, and, while this makes the resulting program as efficient as possible, it does bind the compiler to a specific platform. Because the Alma-0 compiler was developed on a Macintosh platform, but was designed to run on Unix platforms as well, C was chosen as the "machine language".

In the Alma-0 compiler, code generation and the implementation of the AAA, are handled by the C source files a0aaa.c, a0aaa.h, and a0aaaruntime.h. The file a0cprefix.txt provides a template for the output file a.out.c, and contains functions that implement the AAA run-time system, e.g. the log administration system.

3.1.6 The symbol table and other utility functions

Variables, constants, types, and procedures defined in the source file should be remembered for later reference. The *symbol table* takes care of this; a variable can be stored in the symbol table and later on it can be retrieved from the symbol table by searching for its name. The symbol table is used in the semantic analysis phase, as well as in the intermediate code generation phase. There are a number of other utility functions used throughout the compiler, which deal with memory management, string handling and error handling.

The symbol table is implemented by the C source files a0symbol.c and a0symbol.h. Memory management is handled by the C source files a0mem.c and a0mem.h. Strings are handled by the C source files a0string.c and a0string.h, and errors handling is implemented by the C source files a0error.c and a0error.h.

3.1.7 Syntax-directed translation

While the decomposition into phases provides us with a helpful framework, it may seem as if the phases are executed in a serial fashion. This is not true. In fact the phases are intermingled; a few lines are read from the source file, some tokens are recognized, and, when a non-terminal is discovered by the parser, semantic analysis is immediately performed, and, when possible, intermediate code is generated. Only code generation is performed after all the other phases have been executed.

The context-free rules used to specify the syntax, associate with every non-terminal an action to be executed when it is recognized. These actions call the functions that handle semantic analysis and intermediate code generation. This process is called *syntax directed translation* (see [ASU86, chapter 5]).

3.2 Putting it together

3.2.1 Building the compiler

To compile an Alma-0 program, the Alma-0 compiler needs to be built first. Of course this only has to happen once, and is done as follows (see figure 3.1):

1. Flex is run to read the file a0gram.l and generate the C source file lex.yy.c, which contains the C implementation of the lexical analyzer.

2. Bison is run to read the context-free syntax file a0gram.y and generate the C source file y.tab.c, which contains the C implementation of the parser. Bison also writes a verbose log into the file y.output.

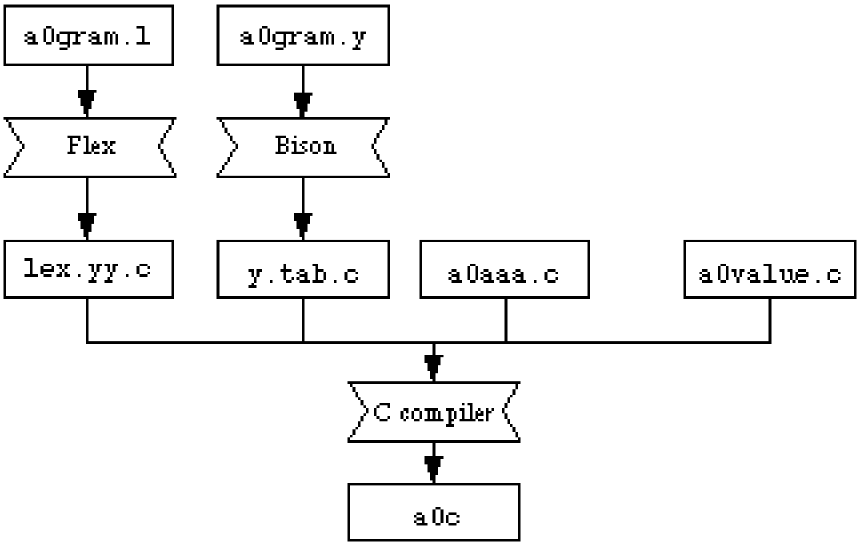


Figure 3.1: Building the Alma-0 compiler

3. The C compiler (CodeWarrior or gcc) is run to read the C source files and generate the Alma-0 compiler executable `a0c`. The C source files generated by Flex and Bison, as well as those written by the compiler implementor (from `a0aaa.c` to `a0value.c`) are read.

On Unix systems, a makefile (`Makefile`) handles these steps, while on Macintosh systems they are handled by a makefile (`Makefile.mac`) and a CodeWarrior project file (`a0c.`μ).

3.2.2 Compiling a program

When the Alma-0 compiler `a0c` has been built, an executable can be built from an Alma-0 source file as follows (see figure 3.2):

1. The Alma-0 compiler `a0c` is run to read the Alma-0 source file (e.g. `queens.a0`), and generate the C file `a.out.c`, which contains the C code generated from the Alma-0 program *and* a copy of the AAA run-time system.

2. The C compiler is run to read the C source file `a.out.c`, and generate the executable (e.g. `a.out`). The executable is stand-alone, i.e. it can be run independently from the Alma-0 system.

3. `a.out` is run to execute the Alma-0 program, e.g. the queens problem.

On Unix systems steps 1 and 2 are automatically handled by the `almac` script, which is installed by running the `INSTALL` script.

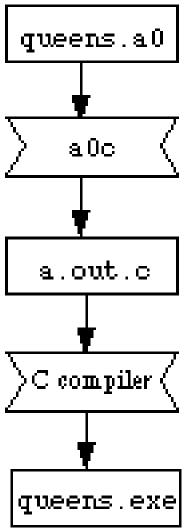


Figure 3.2: Compiling an Alma-0 program

Chapter 4

Lexical and syntax analysis

The syntax of the Alma-0's programming language (see appendix A) was derived from the syntax of Modula-2 as described in appendix B of [Wir85], hereafter referred to as the *original Modula-2 grammar*. Changes were made to allow for the extensions, missing features, and other small changes.

4.1 Lexical analysis

The lexical rules of Alma-0 have been derived from rules 1 to 10 of the original Modula-2 grammar. Only a few changes were made:

- Octal and hexadecimal constants are not recognized.
- The regular expression, which recognizes floating point constants, was changed to require the dot to be followed by at least one digit. This prevents ambiguities in the case of ranges, e.g. before the change `[1..5]` was tokenized as a `REAL`, followed by a dot and an `INTEGER`, instead of an `INTEGER` followed by a double-dot and an `INTEGER`.
- Tokens for new keywords were added, and tokens for unsupported keywords were removed.
- The token <> was provided as an alternative to `#`.

4.2 Syntax analysis

The syntax of Alma-0 has been derived from rules 11 to 91 of the original Modula-2 grammar. To accommodate the **BES** and **SBE** extensions, the distinction between expressions and statements was made less strict:

- At nearly all locations in the original Modula-2 grammar where the non-terminal `expression` was used on the right hand side of a rule, the non-terminal `statement` was put instead.
- All statement types but the assignment and the `RETURN` statement, were moved from the right hand side of the `statement` rule (rule 25 of the syntax in appendix A) to the right hand side of the `factor` rule.
- `expression` was added to the right hand side of the rule for `statement`.

Commentary

4.3c The designator/qualident parse conflict

One of the first problems to surface when implementing the parser, was a parse conflict in the original grammar; The `qualident` and `designator` rules (rules 12 and 14) cause "`a.b`" to be parsable either as a `qualident` or a `designator`.

It seemed impossible to change the grammar to solve this problem. Removing the second alternative from the `designator` rule (rule 14) solves the parse conflict, but makes "`a[5].b`" unparsable. Likewise, removing the second alternative from the `qualident` rule (rule 12) causes "`a.b`" to be parsed as a `designator`, even where a `qualident` is expected.

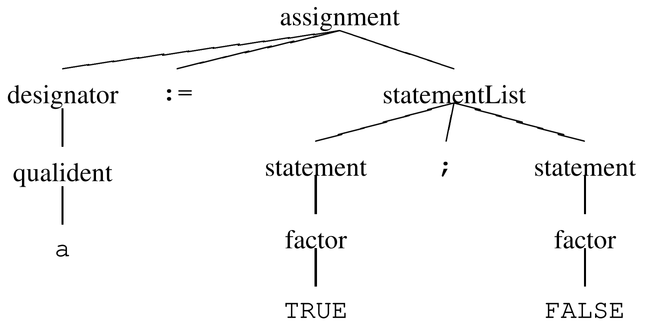
Fortunately Bison handles parse conflicts like this one, known as *shift/reduce conflicts* (see [ASU86, pp. 213–215]), deterministically; it always prefers *shifting* to *reducing*, which in this case means that the correct, second alternative is chosen. To shut off the warning messages that Bison emits when a shift/reduce conflict occurs, the `%expect` command was used.

4.4c Parsing an expression as a statement and vice versa

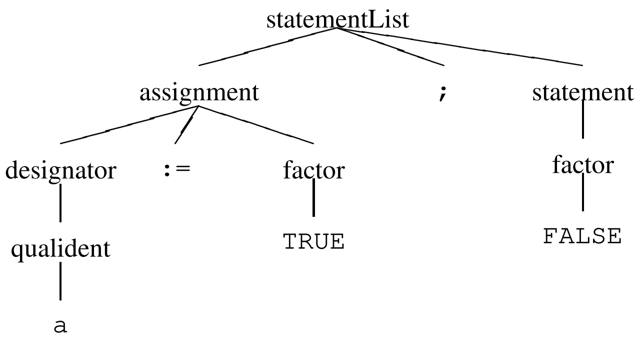
When the grammar was first adapted to allow for the **BES** and the **SBE** extensions, the approach taken was simple. `expression` was added to the right hand side of the `statement` rule and *all* occurrences of `expression` on the right hand side of grammar rules, were changed to `statementList`.

4.4.1c The assignment ambiguity

Unfortunately this caused a parse conflict concerning assignments: the code fragment `a := TRUE; FALSE` could now be parsed either as



or as



We opted for the second interpretation, therefore the syntax rules were changed so that the source of an assignment should be an `expression` instead of a `statementList`.

4.4.2c The parentheses problem

After a while we discovered that the new syntax rules made parentheses mandatory where they should not be. For example, we had to write

```
IF NOT ( FOR i := 1 TO n DO ... END ) THEN ... END
```

where we wanted to write

```
IF NOT FOR i := 1 TO n DO ... END THEN ... END
```

This was caused by the `factor` rule (rule 32), which specified that the operand for the `NOT` operator must be a `factor` and that a `statementList` between parentheses was a `factor`:

```
factor              : NOT factor
                    | '(' statementList ')'
```

The only way to make a `statement` into a `factor` was to put parentheses around it. The first solution to this problem we tried was adding the following rule to the syntax:

```
factor              : statement
```

but this caused a parse conflict concerning assignments similar to the one described above. So, instead the right hand sides of the `statement` rule that described flow control statements, were moved to the right hand side of the `factor` rule, e.g.

```
factor              : WHILE statementList DO statementList END
```

Now every flow control statement could be used directly as `factor` and no parse conflicts would arise. This syntax was the definitive syntax.

Chapter 5

Symbols and symbol tables

During compilation, the compiler needs to keep track of all kinds of named (e.g. types, variables, etc.), and unnamed (e.g. literal values, *instruction sequences*, *temporaries*, etc.) data structures. These data structures are called *symbols*, and the named symbols are stored in the *symbol table* for retrieval by name.

This chapter describes the different kinds of symbols, the way symbols are represented, and the symbol table.

5.1 Different kinds of symbols

Although there are a number of different kinds of symbols, all of them have at least the following features in common:

- a *symbol name*, which is empty for unnamed symbols.
- a variable, which specified the kind of symbol it is, called the *kind identifier*.
- a link to the next symbol in the symbol table.

The symbol name and the kind identifiers are orthogonal, i.e. there are named as well as unnamed type symbols, named as well as unnamed variable symbols, etc. Sometimes, two different, but similar, language features are represented by one kind identifier, with the difference being the presence, or absence, of a name. For example, constants and literal values are both considered *values*; the first a named value, the second an unnamed value.

We can distinguish *type symbols*, *value symbols*, *code symbols*, *procedure symbols*, and *sameas symbols*. The next sections described these symbols and their specific features.

5.1.1 The type symbol

The type symbol represent an Alma-0 type. We can define the following groups of types:

- The *built in types* are `BOOLEAN`, `CHAR`, `INTEGER`, and `REAL`, as well as a number of internal types.
- The *basic types* are the built in types, enumeration types, and range types, as well as the *reference types* used internally.

- The *non-basic types* are the types that are not basic types, i.e. record types and array types.

Like symbols, there are different kinds of types, with different specific features. A *type kind identifier* distinguished between the following kinds:

Built in types `BOOLEAN`, `CHAR`, `INTEGER`, and `REAL`. There are AAA instructions to handle values of these types directly.

Enumeration types The specific feature is a list of the constants of the enumeration type.

Range types The specific features are the upper and lower bound, as well as the *base type* of the range type.

Record types The specific feature is list of the fields of the record type.

Array types The specific features are the key and value types of the array type.

Reference types These are used internally to represent pointers. The specific feature is the type of the value pointed to, which may not be another reference type.

5.1.2 The value symbol

The value symbol represents the following objects:

Literal values Integer and floating point numbers, as well as character and string values, without a name.

Constants Literal values that have been given a name by a `CONST` declaration.

Variables A memory location declared and given a name by a `VAR` declaration.

Temporaries A memory location used to hold a temporary value, e.g. the result of an addition, the return value of a procedure. Temporaries are the unnamed counterpart to variables.

Specific features for the value symbols are the type of the value, and an AAA operand that represents the actual value. The different objects represented by the value symbol can be distinguished by the absence or presence of a name, by the type, and by the operand.

5.1.3 The code symbol

The code symbol, which is always unnamed, represents a sequence of instructions, also called *code*. The first code generated, e.g. code for an addition, represent short sequences, but as constructs higher in the hierarchy are translated, e.g. a `WHILE` statement, the sequences get longer and longer. Eventually one code symbol represent the entire translation of the body of a procedure, or of the main code. We can define the following kinds of code, depending upon their result:

Void code Code that has no result, e.g. a flow control statement, or a call to a procedure that returns no value.

Computational code Code that has a result, e.g. an addition, or a call to a procedure that returns a value. A value symbol represents the result.

Conditional code Code that evaluates a boolean expression, e.g. an `AND` operation, or a call to the `KNOWN` procedure. When the boolean expression evaluates the `TRUE`, execution continues after the code. When the boolean expression evaluates to `FALSE`, execution continues at location indicated by the *false continuation label*[1].

The specific features of the code symbol are:

- A pointer to the sequence of AAA instructions.
- A pointer to the value that represents the result, when the code is computational. Special values indicate void code and conditional code.
- The false continuation label, when the code is conditional.

A sequence of AAA instructions is represented by a singly linked list of *instruction records*. Each instruction record contains an opcode, three operands, and a pointer to the next instruction in the list.

5.1.4 The procedure symbol

The procedure symbol, sometimes abbreviated to *proc symbol*, represents an Alma-0 procedure. It is always named, and has the following specific features:

- A list of formal parameters.
- A return type.
- An value symbol, which represents the return value.
- A pointer to a code symbol, which represents the AAA instructions generated for the procedure body.
- A label that is placed at the start of the procedure body, the *start label*.
- A label that is placed at the end of the procedure body, the *end label*.

5.1.5 The sameas symbol

The sameas symbol represents a symbol that is identical to another symbol. A symbol can have only one name, therefore when two identifiers refer to the same symbol, a sameas symbol is created; one identifier refers directly to the symbol, the other identifier refers to the sameas symbol, which contains a pointer to the original symbol. When a pointer to a sameas symbol is encountered, it is immediately substituted for a pointer to the original symbol. All this is handled automatically by the symbol table handling functions.

5.2 Symbol representation

Because the different kinds of symbols have common, as well as specific features, they lends themselves well to an object-oriented design. However, the C language does not provide the programmer with any tools for object-oriented programming, so that the eventual design can only be described as *pseudo object-oriented* (see section 5.4 for a description of the problems encountered before finally settling for this design). The following C types represent symbols:

[1] A label is an identifier, which is used to indicate the target of a branch or jump instruction. A `LAB` instruction, with the label as its only operand, is inserted into the instruction sequence at the location where the jump should be made to.

```
typedef struct symbol {
    struct symbol   *next_symbol;
    enum symbol_kind {
        kind_type = 1,
        kind_value,
        kind_code,
        kind_proc,
        kind_sameas
    }              kind;
    char            *name;
} symbol_t;

typedef struct type {
    symbol_t        sym;
    ... data specific for a type ...
} type_t;

typedef struct value {
    symbol_t        sym;
    ... data specific for a value ...
} value_t;

... idem dito for code, procedure and sameas ...
```

The auxiliary function `symbol_t *symbol_new(enum symbol_kind kind)` allocates a new symbol of the right kind, and of the right size. When requesting a value, `symbol_new()` does something like this:

```
symbol_t *newsym = (value_t *) malloc(sizeof(value_t));
newsym->next_symbol = NULL;
newsym->kind = kind_value;
newsym->name = NULL;
return newsym;
```

When a symbol of an unknown kind is encountered, the kind can easily be determined by checking the `kind` field:

```
if(sym->kind == kind_value) {
    value_t    *val = (value_t *) sym;
    ... val points to a value ...
}
```

5.3 The symbol table

When a named symbol is created, e.g. when a variable is declared, it is added to the *current symbol table*. One symbol table can hold all kinds of symbols, although the code symbols, which are always unnamed, are never added to a symbol table, and procedures are never added to a local symbol table, because Alma-0 does not support nested procedures.

A symbol table is represented by a singly linked list. New symbols are added to the front of the list that represents the current symbol table. Normally the *global symbol table*, which contains all global symbols, is the current symbol table, but during compilation of a procedure, its *local symbol table* is the current symbol table.

When a symbol is looked up, the local symbol table, if present, is traversed first. If the symbol was not found there, the global symbol table is traversed too. If the symbol is neither found there, a “symbol not found” error is generated. If a symbol has been found, the kind identifier can be checked to determine the kind of the symbol.

Commentary

5.4c Implementing the object-oriented design in C

The symbol management system lends itself well to an object oriented design; there is an *is-a* relationship between symbols and the different kinds of symbols, e.g. a type symbol *is a* symbol. Unfortunately the C programming language does not provide for a way to express this relationship in a natural way, unlike languages like C++ [ES90] and Java [GJS96]. A way had to be found to implement this design nonetheless.

5.4.1c First approach

First one type was declared, a `struct` (called `lobj_t`, short for ”language object type”) which contained a union for the specific info for each kind, and a field to distinguish between the different kinds:

```
struct value_info {
    ... data specific for a value ...
};

struct type_info {
    ... data specific for a type ...
};

typedef struct lobj {
    enum {
        kind_value = 1,
        kind_type,
        ...
    }                   kind;
    union {
        struct value_info vi;
        struct type_info  ti;
        ...
    }                   u;
} lobj_t;
```

Note that the `lobj_t` structure had no `name` field. Instead, the symbol table contained a list of (name,pointer) pairs that associated a name with a pointer to a `lobj_t` structure. When two names referred to the same thing the corresponding symbol table entries pointed to the same `lobj_t`. The sameas symbol was not necessary using this approach.

This design had the advantage that no type casting was necessary to use a `lobj_t` as a specific kind of symbol, but this also meant that the C compiler's type checking mechanism was bypassed completely, and that a function had to check the kind identifier, to be sure it was passed a symbol of the correct kind. The absence of a name field also made debugging harder.

5.4.2c Second approach

Because `lobj_ts` were used extensively throughout the compiler, the bypassing of the type checking mechanism led to an increase in the number of unnoticed programming errors. A new symbol system was designed in which the `lobj_t struct` kept pointers to the kind-specific data instead of the data itself:

```
struct value_t {
    ... data specific for a value ...
};

struct type_t {
    ... data specific for a value ...
};

typedef struct lobj {
    enum {
        kind_value = 1,
        kind_type,
        ...
    }                   kind;
    union {
        struct value_t    *vi;
        struct type_t     *ti;
        ...
    }                   u;
} lobj_t;
```

This design allowed functions to specify, in their prototype, what kind of symbol they expected, instead of having to check the kind identifier explicitly, thereby allowing the compiler to find programming errors earlier on. However, more memory was now allocated for each symbol; one `lobj_t struct` and a C `struct` containing kind-specific data. Furthermore, there was still no name field.

5.4.3c Third and final approach

While this second approach proved quite successful (and indeed lasted for quite some time before being replaced), the disadvantages finally led to the design described in section 5.2. The `lobj_t` type was renamed to `symbol_t`, the generic data and the kind-specific data were merged into one C `struct` and a name field was added.

Adding a `name` did introduce two new problems; what should happen when two identifiers refer to the same symbol, and, how should unnamed symbols be handled? Unnamed symbols are handled simply by setting their name field to `NULL`, and the case of two identifiers referring to the same symbol is handled by the sameas symbol.

Chapter 6

Semantic analysis

Semantic analysis actually compromises a lot of different activities, a.o. type analysis, constant folding, and variable allocation, all of which are described in this chapter.

6.1 Type analysis

As has been described in chapter 3, type analysis concerns itself with checking if the program being compiled contains no typing errors.

6.1.1 Type compatibility

The primary notion here is that of *type compatibility*[1], also called *type equivalence*. In Alma-0 two types `s` and `t` are *compatible*, when at least one of the following is true:

- `s = t`
- `s` is a range with base type `t`
- `t` is a range with base type `s`
- `s` and `t` are both ranges with the same base type

This kind of type equivalence is called *name equivalence* [ASU86, pp. 352–359]. The types of expressions are checked in the following cases:

- Binary operations may only be performed on two values of compatible types, e.g. a `REAL` value can be added to a `REAL` value, but not to an `INTEGER` value. Assignment can be seen as a binary operation in this respect, i.e. only a value of a compatible type may be assigned to a variable of a certain type.
- Some operations may not be performed on all types, e.g. the binary minus may be applied to a `INTEGER` value, but not to a `BOOLEAN` value.
- Only a value of a *compatible* type may be used as the actual parameter for a procedure's call-by-value parameter. This also applies to a call-by-mixed-form parameter, when it behaves as a call-by-value parameter.

[1] Alma-0 doesn't know the separate notion of *assignment compatibility* because there is no `CARDINAL` type

- Only a value of an *identical* type may be used as the actual parameter for a procedure's call-by-variable parameter. This also applies to a call-by-mixed-form parameter, when it behaves as a call-by-variable parameter.

6.1.2 Type representation

Type symbols are represented by the following C types:

```
struct enum_info {
    struct string_list_elt  *vals;
};

struct range_info {
    type_t              *basetype;
    value_t             *from, *to;
};

struct record_field {
    struct record_field *next;
    char                *name;
    aaaword_t           offset;
    type_t              *type;
};

struct record_info {
    struct record_field *fields;
};

struct array_info {
    type_t              *key_type, *val_type;
};

struct reference_info {
    type_t              *ref_type;
};

struct type {
    symbol_t            sym;
    enum {
        kind_builtin = 1,
        kind_enum,
        kind_range,
        kind_record,
        kind_array,
        kind_reference
    }                   kind;
    aaaword_t           size;
    union {
        struct enum_info        enumeration;
        struct range_info       range;
        struct record_info      record;
        struct array_info       array;
        struct reference_info   reference;
```

```
    }                               u;
} type_t;
```

- The `kind` field identifies the kind of type.
- The `size` field contains the total size in bytes of the type.
- The `u` field is a union of records, containing data specific for a certain kind of type.

Only one `type_t` structure is allocated for each type, and the base type of a range, is available through the `basetype` field, making it easy to determine type compatibility.

6.1.3 Type coercion

Although a function could be made to determine type compatibility, a different approach was taken. Instead of calling this hypothetical function to check if two types are *compatible*, a call is made to the function `code_coerce()`, and afterwards a check is made to see if the types are *identical*. If the types were compatible, but not identical, the function `code_coerce()` would have coerced the type of one value into the type of the other, thereby making them identical.

Coercion is the implicit conversion, by the compiler, of a value of one type, into a value of another type. Usually the new value conveys the same information as the old value, e.g. in a lot of programming languages, but not Alma-0, integer values are automatically coerced into floating point values, although the reverse conversion is usually not performed automatically, because information would be lost.

The function `code_coerce()` takes two arguments, the first a pointer to a code symbol pointer, the second a pointer to the type requested, and coerces the result of the code into the requested type, if possible. The function `code_coerce()` does not coerce values, but before a value symbol is used in an expression, where coercion could apply, it is converted into a computational code symbol, whose result is the original value. Therefore the function `code_coerce()` can also handle these cases.

The following code fragment is an example of how type compatibility could be checked:

```
code_coerce(&code_to_check, type_to_check_against);
if(code_to_check->result->type == type_to_check_against) {
    ... types are compatible ...
} else {
    ... types are not compatible ...
}
```

The coercions performed, when needed, by the function `code_coerce()`, are:

- A value of a type that is compatible with, but not identical to, another type can be coerced into a value of that type.
- A reference to a value of a type that is compatible with, but not identical to, another type can be coerced into a reference to a value of that type.
- A reference to a value of a basic type can be coerced into the value referenced (the value is *dereferenced*).
- Computational code that returns a boolean value, can be coerced into conditional code.
- Conditional code can be coerced into computational code that returns a boolean value.

- Conditional code can be coerced into void code. This coercion implements the **BES** extension.
- Void code can be coerced into condition code. This coercion implements the **SBE** extension.
- A constant of type `STRING` can be coerced into an array of type `CHAR`, provided the array is large enough to hold the complete string.

6.2 Values

In the Alma-0 compiler there is a single abstraction, the value, to represent literal values, constants, variables, and temporaries. The following section describe values in general, as well as the details of the different kinds of values.

6.2.1 Value representation

Values are represented by the following C type:

```
typedef struct value {
    symbol_t    sym;
    type_t      *type;
    operand_t   oper;
} value_t;
```

- The `type` field is a pointer to the type of the value represented by the `oper` field.
- The `oper` field represents the actual value as an AAA operand (see section 8.1.1 for the representation of AAA operands).

The name, `type`, and `oper` fields determine the kind of value we're dealing with:

- When the type is `INTEGER` and the addressing mode of the operand is `immword`, the `value_t` represents an immutable `INTEGER` value, i.e. a literal value or a constant. If the value is named, it is a constant, otherwise it is a literal value.
- When the type is reference-to-`INTEGER` and the addressing mode of the operand is `regnval`, the `value_t` represents the *address* of an `INTEGER` value in memory, i.e. an lvalue. If the value is named, it is a variable, otherwise it is a field of a record, or an element of an array.
- When the type is `INTEGER` and the addressing mode of the operand is `indword`, the `value_t` represents an `INTEGER` value in memory, e.g. a dereferenced lvalue or a temporary.

Note the difference between the last two `value_ts`; one represents the address of an lvalue, the other the value itself. An address of a variable can be coerced into the value by dereferencing the value.

Only references to basic types can be dereferenced, because indirect addressing modes only exists for the basic types. References to variables of non-basic types (arrays and records) are not dereferenced; only its elements are (provided they are of a basic type), after the references have been used to calculate the addresses of the elements.

6.2.2 Constant folding

The Alma-0 compiler performs *constant folding* when possible. Constant folding is the process of replacing expressions, which can be evaluated at compile-time, by their result. For example, the expression “`30+2*4`” can be replaced by the expression “`38`”, thereby generating code that runs quicker.

Apart from being useful by optimizing the generated code, constant folding is also *necessary* at some points in the Alma-0 language; the dimensions of an array should be known at compile-time, which means that the expressions in a declaration like

```
VAR
    info: ARRAY [MINVAL-1 .. MAXVAL*2] OF INTEGER;
```

should be evaluated at compile-time. It turns out that mandatory constant folding only applies to arithmetical operators, and not to logical or comparison operators. Therefore the Alma-0 compiler only performs constant folding where it concerns arithmetical operators, but the concept could easily be extended to also include the logical and comparison operators.

Earlier versions of the Alma-0 compiler also performed constant folding where it concerned address calculations. For example, the address of `info[MINVAL]` can be determined at compile-time. However, these optimizations were removed, because they made the implementation more complex, which would complicate further development on the compiler by someone else.

6.2.3 Variable allocation

When a variable is declared, memory is allocated for it. If the variable is a global variable, the memory is allocated in main memory, otherwise it is allocated on the stack. The allocation strategy used is quite simple; a pointer keeps track of the amount of memory allocated already, and is incremented each time a variable (or temporary) is allocated. No attempt is made to interleave variables. At the end, this pointer tells the compiler how much main memory, or stack memory, to reserve for global, or local, variables.

Registers are not used to hold variables, as it would require the log administration functions to specifically remember if a variable was stored in memory or in a register. This would increase the overhead of the log administration system.

The **EQ** and **KNOWN** extensions require the run-time system to keep track of whether a variable is initialized or uninitialized. To this end, with every lvalue of a basic type, a flag is associated. The flag is initially false (to denote that the variable is uninitialized), and is set to true (to denote that the variable is initialized) when a value is assigned to the variable. When the variable is dereferenced (and its value is about to be used), the flag is checked and a run-time error is generated when it is false. The only exception to this rule is the equality operator =, which only generates a run-time error if both sides of the equality are unknown. Note that the target of an assignment is *not* dereferenced.

This flag can be represented by a byte, which is stored with every lvalue. This makes all lvalues at least one byte larger, and, because of alignment, types can become up to 8 bytes (the alignment factor on the Sparc Solaris platform) larger. Although a bit would suffice to represent the flag, storing it without reducing the number of bits reserved for characters, integers, and reals, requires a byte. One can imagine storing all the bits for one record or for a number of contiguous array elements in one byte, but implementing this would be complicated.

6.2.4 Temporaries

Temporaries are temporary, unnamed, variables used to hold intermediate values of calculations. For example, the result of the calculation “`i+1`” is stored in a temporary before being used in an assignment, in another expression, or as an actual parameter.

Technically, temporaries are variables that have already been dereferenced for the convenience of the compiler implementor; it saves calling `code_coerce()` to dereference them. Because only references to basic types can be dereferenced, temporaries can only be allocated for basic types.

Temporaries share the allocation strategy of regular variables. Although it is possible to employ a different strategy (e.g. registers could be used), it seemed very complicated to implement, and it was therefore not done.

6.3 Procedures

6.3.1 Procedure representation

Procedure symbols represent Alma-0 procedures, and the corresponding C types are:

```
typedef enum {
    call_val = 1,
    call_var,
    call_mix
} callmech_t;

struct formal_param {
    struct formal_param *next;
    char                *name;
    callmech_t          callmech;
    type_t              *type;
};

struct proc {
    symbol_t            sym;
    struct formal_param *formal_params;
    type_t              *return_type;
    value_t             *return_value;
    code_t              *code;
    aaaword_t           start_label,
                        end_label;
} proc_t;
```

- The `formal_params` field points to a linked list that represents the formal parameters.
- The `return_type` field points to the return type of the procedure, or contains `NULL` when the procedure has no return value.
- The `return_value` field represents the location in the stack frame where the return value should be stored. This field is used when a `RETURN` statement is translated.
- The `code` field points to the code of the procedure body.
- The `start_label` and `end_label` fields contain the start label and the end label.

Commentary

6.4c Why Alma-0 has no CARDINAL type

Modula-2 adds the CARDINAL to the types borrowed from Pascal [Wir76], and at the same type strengthens the definition of type compatibility to the definition given in section 6.1.1. This and a few other definitions in [Wir85] lead to the following contradiction:

The types CARDINAL and INTEGER are not compatible.
All positive non-floating-point literal values are of CARDINAL type.
The unary minus can only be used on operands of INTEGER type.

The construct -5 is illegal.

To circumvent this problem a lot of special cases had to built into the type checking routines, e.g. a CARDINAL constant smaller than MAXINT can automatically be coerced into an INTEGER constant. Because this made the compiler design very complicated, and distracted from the actual purpose of the project, support for the CARDINAL type was removed from the compiler.

Chapter 7

Intermediate code generation

The intermediate code generation phase is a very interesting phase; during this phase the language constructs are actually translated into sequences of instructions. In the case of the Alma-0 compiler, the target platform is the AAA described in chapters 2 and 8.

7.1 Code representation

A code symbol is represented by the following C type:

```
typedef struct code {
    symbol_t    sym;
    value_t     *result;
    aaaword_t   false_lab;
    instr_t     *instr;
} code_t;
```

- The `result` field points to the value that represents the result, if the code is computational. If the code is void, the `result` field points to the special value `value_void`, and if the code is conditional, the `result` field points to the special value `value_cond`.
- The `false_lab` field contains the false continuation label, if the code is conditional.
- The `instr` field points to a linked list of AAA instructions.

7.2 Translating language constructs into code

Because of the syntax directed translation technique used, an Alma-0 language construct is translated into AAA instructions, as soon as it has been recognized by the parser. The bottom-up parsing strategy ensures code has already been generated for the language constructs contained by the current construct, i.e. those language constructs that are its descendants in the abstract syntax tree. This means that the result of computational code can be used, and that conditional code, and its false continuation label, can be correctly placed to get the correct flow of control.

The translation of most language constructs is obvious (see [ASU86] and [Wir96] for examples), and therefore we will only discuss those translations that deal with Alma-0's extensions.

7.2.1 Pseudo code

Because the actual instruction sequences generated can be quite long, we will use pseudo code to illustrate the idea. Check the source files of the compiler (in particular the files `a0code.c`, `a0code_ops.c`, and `a0code_flow.c`) for the actual AAA instructions generated. The following language constructs are used in the pseudo code:

- `create_frame_and_save_values(<frame-type>, <registers>)` is a "function", which creates room on the stack for the specified type of frame, and stores the values of the specified registers in the frame. The base address of the new frame is returned.
- `(<registers>) := restore_values(<frame-type> <frame-base-address>)` is a "function", which restores the values of the specified registers from the specified type of frame.
- `destroy_frame(<frame-type>)` is a "function", which destroys the specified type of frame.
- `(<registers>) := restore_values_and_destroy_frame(<frame-type>, <frame-base-address>)` is a "function", which restores the values of the specified registers, and destroys the specified type of frame.
- `x := y` is identical to the AAA instruction "`ADD x, y, 0`".
- `IF x op y THEN a ELSE b END` is identical to the AAA instructions:

  ```
      Bop x, y, true_lab;
      b;
      BRA continue_lab;
  true_lab:
      a;
  continue_lab;
  ```

  where `Bop` is the comparison-and-branch instruction, which performs the correct comparison, e.g. `Bop = BLT`, if `op = '<'`.
- Assignments which are implemented by the recording version of an instructions such as `ADD`, are marked by "`(* recording version *)`".
- Although the instruction `REWINDLOG` is never explicitly used in the pseudo code fragments, nearly every assignment to the `LP` register is actually implemented by the `REWINDLOG` instruction, which takes care of cleaning up logs which would otherwise remain allocated. Only for the correct translation of the `FORALL` statement, is direct assignment to the `LP` register needed. See the compiler source files for details.

7.2.2 BES

When a boolean expression (`be`) is used as a statement (`s`), the following code is generated:

```
    be.instr;
    BRA true_lab;

be.false_lab:
    FAIL;

true_lab:
```

- If the boolean expression evaluates to TRUE, execution continues normally, after the label true_lab.
- If the boolean expression evaluates to FALSE, the FAIL instruction is executed, causing a jump to the last failure point.

7.2.3 SBE

When a list of statements (s) is used as a boolean expression (be), the following code is generated:

```
    BP := create_frame_and_save_values(SBE_FRAME, LP, BP, EP);
    temp := BP;
    ONFAIL fail_lab;

    s;

    (LP, BP, EP) := restore_values_and_destroy_frame(SBE_FRAME, temp)
    BRA succeed_lab;

fail_lab:
    (LP, BP, EP) := restore_values_and_destroy_frame(SBE_FRAME, BP)
    BRA be.false_lab;

succeed_lab:
```

- If s succeeds, the saved values are restored, and execution continues normally. Because BP may point to a frame created during the execution of s, temp is used as the pointer to the frame instead.
- If s fails, the saved values are restored, and a jump is made to the new false continuation label be.false_lab. Because this is the failure handler installed at the beginning, the register BP will now point to the correct frame.

7.2.4 ORELSE

The statement

```
EITHER s ORELSE t ORELSE u END;
```

is translated into:

```
    BP := create_frame_and_save_values(ORELSE_FRAME, BP, EP)
    CREATELOG;
    ONFAIL second_branch_lab;
    s;
    BRA continue_lab;

second_branch_lab:
    REPLAYLOG;
    EP := restore_values(ORELSE_FRAME, BP)
    CREATELOG;
    ONFAIL final_branch_lab;
    t;
```

```
    BRA continue_lab;

final_branch_lab:
    REPLAYLOG;
    EP := restore_values(ORELSE_FRAME, BP)
    BP := restore_values_and_destroy_frame(ORELSE_FRAME, BP);
    u;

continue_lab:
```

- A failure handler is installed and a log is created for all but the last branch.
- If the execution of a branch (but not the last one) fails, the log is replayed, and the next branch is tried.
- If execution of the last branch fails, no special action should be performed by the `ORELSE` statement, and therefore no failure handler is installed and no log is created, for the last branch.

7.2.5 SOME

The statement

```
SOME i := a TO b BY incr DO s END:
```

is translated into:

```
    IF a > b THEN
        FAIL;
    ELSIF a = b THEN
        s;
    ELSIF a < b THEN
        BP := create_frame_and_save_values(SOME_FRAME, BP, EP);
        CREATELOG;
        ONFAIL fail_lab;
        BRA loopbody_lab;

fail_lab:
        REPLAYLOG;
        EP := restore_values(SOME_FRAME, BP);

        i := i + incr;  (* recording version *)
        IF i < b THEN
            CREATELOG;
            ONFAIL fail_lab;
        ELSE
            BP := restore_values_and_destroy_frame(SOME_FRAME, BP);
        END

loopbody_lab:
        s;
    END;
```

- When `incr < 0`, every occurrence of the < should be replaced by >.

- The `SOME` statement behaves like an iterated `ORELSE` statement; if the loop is unrolled, the instructions will be similar to those generated for the `ORELSE` statement.

7.2.6 COMMIT

The statement

```
COMMIT s END
```

is translated into:

```
    savesp := SP;
    savebp := BP;
    savelp := LP;

    s;

    LP := savelp;
    BP := savebp;
    SP := savesp;
```

- After execution of `s`, any failure handlers installed by `s` or logs created by `s`, are deleted by restoring the old values of LP, BP and SP.

7.2.7 FORALL

The statement

```
FORALL s DO t END
```

is translated into:

```
    BP := create_frame_and_store_values(FORALL_FRAME, LP, BP);
    saveorigbp := BP;
    CREATELOG;
    ONFAIL forall_done_lab;

    s;

    savesp := SP;
    savebp := BP;
    savelp := LP;

    (LP, BP) := restore_values(FORALL_FRAME, saveorigbp);
    t;

    LP := savelp;
    BP := savebp;
    SP := savesp;
```

```
    FAIL;

forall_done_lab:
    REPLAYLOG;
    BP := restore_values_and_destroy_frame(FORALL_FRAME, BP);
```

- Before `t` is executed, the context active before the `FORALL` statement is restored. This prevents the assignment in `t` from being undone when backtracking takes place in `s`.
- An implicit `COMMIT` statement surrounds the `DO` part of the `FORALL` statement, to delete any choice points created during execution of `t`.
- The `FAIL` instruction causes a jump to the last failure handler installed in `s`. When no more failure handlers are left in `s`, execution will continue at `forall_done_lab`. This approach is similar to that of the failure-driven loop.

7.2.8 EQ

The code generated for the equality operator depends upon the objects being compared:

- When two non-lvalues are being compared, the code generated is no different from that generated for the other comparison operators.
- When two lvalues (`lhs` and `rhs`) are being compared, the following code is generated:

```
IF lhs.initialized THEN
    IF rhs.initialized THEN
        IF lhs.initialized <> rhs.initialized THEN
            BRA false_lab;
        END
    ELSE
        rhs := lhs; (* recording version *)
    END
ELSE
    IF rhs.initialized THEN
        lhs := rhs; (* recording version *)
    ELSE
        generate_runtime_error;
    END
END
```

- When an lvalue and a non-lvalue are being compared, the code generated is similar to that above, but the initialized flag of the non-lvalue expression is not checked.

7.2.9 MIX

The implementation of the call-by-mixed-form mechanism mostly resembles that of the pass-by-variable mechanism:

- The callee expects a reference to an lvalue, as its formal parameter.

- When the actual parameter is an lvalue, a reference to it is passed to the callee.
- When the actual parameter is not an lvalue, a temporary is created, the value is assigned to the temporary, and a reference to the temporary is passed instead. What happens is similar to rewriting

```
    PROCEDURE max(MIX a: INTEGER):FORWARD;
    BEGIN
        max(x+5);
    END
```

  as

```
    PROCEDURE max(VAR a: INTEGER):FORWARD;
    VAR
        temp:   INTEGER;
    BEGIN
        temp := x+5;
        max(temp);
    END;
```

7.2.10 KNOWN

The expression `KNOWN(x)` is translated into:

```
IF NOT x.initialized THEN
    BRA known.false_lab;
END;
```

- Because of the bottom-up parsing strategy, the information of the individual variables in `x` is no longer available when `KNOWN(x)` is translated; code, which evaluates the expression, has been generated instead. Therefore it is impossible to check the variables in the expression. Therefore this alternative implementation was chosen because the programmer can easily rewrite complicated expressions into a number of applications of the `KNOWN` procedure.

7.2.11 Procedure call

A procedure call in the AAA is a handled slightly differently from a procedure call in a classic virtual machine. The procedure call `proc;` translates to:

```
push_actual_parameters;
EP := create_frame_and_save_values(PROCCALL_FRAME, EP, SP);
JSR proc.label;
(EP, S1) := restore_values(PROCCALL_FRAME, EP);
IF S1 < BP THEN
    destroy_frame(PROCCALL_FRAME);
END;
```

- If a choice point was created in the callee, execution may, at a later point, continue in the body of the procedure. When that happens, its local variables should be accessible and should have the values they had the first time round. Therefore the stack frame is not destroyed, if the failure frame register is equal to or greater than the stack pointer.

Commentary

7.3c Order of code generation

An important characteristic of a compiler, is the *order* in which code is generated. This characteristic is closely intermingled with the parsing strategy employed (bottom-up or top-down) and the temporary allocation strategy.

7.3.1c Immediate code emission

A simple code emission strategy is the one used in [Wir96]. A top-down predictive parser is used, which makes it possible to emit every instruction directly to the output stream during parsing. The following pseudo-code demonstrates this technique. Note that the procedures `parseBooleanExpression` and `parseStatementSequence` also emit their instructions directly into the output stream.

```
PROCEDURE parseWhileStatement;
BEGIN
    loopLabel := getNextLabel;
    skipLabel := getNextLabel;
    match('WHILE');
    emit(LAB, loopLable);
    parseBooleanExpression;
    emit(BRA_ON_FALSE, skipLabel);
    match('DO');
    parseStatementSequence;
    match('END');
    emit(BRA, loopLabel);
    emit(LAB, skipLabel);
END parseWhileStatement;
```

This code emission strategy makes it possible to have a simple temporary allocation strategy. For example, a bitmask of the available registers can be kept. When a temporary is requested, the first register available according the bitmask is allocated, and the bit corresponding to the register is set. When the temporary is deallocated, the corresponding bit can simply be cleared, allowing the register to be used again.

7.3.2c Concatenation of instruction sequences

Unfortunately, the Bison parser generator generates a bottom-up parser, preventing us from employing this strategy; when the action for a language construct is executed the actions for the construct in it have already been executed (e.g. `actionWhileStatement` is executed *after* `actionBooleanExpression` and `actionStatementSequence` have been executed). Therefore, another approach is used; every action returns a pointer to the list of instructions it has created, and higher level constructs concatenate these lists to form larger lists, e.g.

```
PROCEDURE actionWhileStatement(
    booleanExpressionInstructions,
    statementSequenceInstructions : instructions):instructions;
BEGIN
    loopLabel := getNextLabel;
    skipLabel := getNextLabel;
    RETURN concatenateInstructions(
        instruction(LAB, loopLabel),
        booleanExpressionInstructions,
        instruction(BRA_ON_FALSE, skipLabel),
        statementSequenceInstructions,
        instruction(BRA, loopLabel),
        instruction(LAB, skipLabel));
END actionWhileStatement;
```

Using this non-linear instruction emission strategy prevents us from using the simple temporary value allocation strategy described above. If the action `actionWhileStatement` were to allocate a temporary according to this strategy it might get appointed a memory location, which was previously allocated to a temporary used by `actionBooleanExpression`. If instructions using this new temporary were inserted *before* and *after* the instructions generated by `actionBooleanExpression` (i.e. `booleanExpressionInstructions`), the lifetimes of the two temporaries would overlap. Because the two temporaries use the same memory location, their values would become corrupted.

The actual problem is the deallocation that happens too soon; `actionBooleanExpression` deallocates the temporary, which allows `actionWhileStatement` to allocate a temporary at the same memory location. Therefore the temporary allocation strategy of the Alma-0 compiler does not allow the deallocation of temporaries. As registers would fill up very quickly using this approach, temporaries are allocated the same way regular variables are; in main memory, or on the stack.

Chapter 8

Implementation of the AAA

Although code generation and the run-time system are actually two separate subjects, this chapter deals with both, because together they make up the implementation of the AAA. Code generation takes care of translating the AAA instructions into C statements, and the run-time system helps these C statements perform their duties.

8.1 Code generation

8.1.1 Instruction representation

In the implementation of the AAA, addressing modes, operands, opcodes, and instructions are represented by the following C types:

```
typedef enum opcode {
    op_recording = 0x8000,
    op_nop = 0,
    op_add, op_sub, op_mul, op_div, op_mod, op_chk,
    op_move, op_clear,
    op_beq, op_bne, op_blt, op_bge, op_bgt, op_ble, op_bra,
    op_jsr, op_rts, op_onfail, op_fail, op_lab,
    op_createlog, op_replaylog, op_rewindlog,
    op_read, op_write, op_writeln, op_comment
} opcode_t;

typedef enum addrmode {
    am_regnval = 1,
    am_indbyte, am_indword, am_indfloat, am_indstring,
    am_immbyte, am_immword, am_immfloat, am_immstring
} addrmode_t;

typedef struct operand {
    addrmode_t      addrmode;
    union {
        struct {
            aaareg_t    reg;
            aaaword_t   val;
```

```
        }            regnval;
        aaabyte_t    byte_val;
        aaaword_t    word_val;
        aaafloat_t   float_val;
        aaastring_t  string_val;
    } u;
} operand_t;

typedef struct instr {
    struct instr    *next;
    opcode_t        opcode;
    operand_t       operands[3];
} instr_t;
```

- The `op_recording` opcode is a value that can be added to one of the other opcodes, to turn it into its recording versions.

- The `regnval` field of the `operand_t` type is used when the addressing mode is either `regnval`, `indbyte`, `indword`, or `indfloat`.

- The `next` field of the `instr_t` type points to the next instruction in the sequence. This does prevent one `instr_t` record from being part of more than one instruction sequence.

8.1.2 Code selection

Selecting the right C statements to translate the AAA instructions is done according to a straight-forward scheme. First the three operands are translated into fragments of C code, depending on the addressing mode:

Addressing mode	example C code
`regnval`	`R4`
`indbyte`	`(* (aaabyte_t *) mem+R5-27)`
`indword`	`(* (aaaword_t *) mem+R4+56)`
`indfloat`	`(* (aaafloat_t *) mem+76)`
`immbyte`	`'a'`
`immword`	`56`
`immfloat`	`50.005000`
`immstring`	`"I\'m a string."`

The C fragments generated for the operands (abbreviated to `a`, `b` and `c`) are used to generate the C statement that corresponds to the opcode:

Opcode	example C code
`ADD`	`a = b + c;`
`ADD_recording`	`log_record(&a, sizeof(a)); a = b + c;`
`MOVE`	`memcpy(mem[a], mem[b], c);`

`CHK`	`if(a < b && a > c) {`
	`    fprintf("CHK instruction failed; value out of bounds}\n");`
	`    exit(1);`
	`}`

`BEQ`	`if(a == b) goto labc;`
`JSR`	`if(setjmp(* (jmp_buf *) mem+REP) == 0) goto labc;`
`RTS`	`longjmp(* (jmp_buf *) mem+REP, 1);`
`ONFAIL`	`if(setjmp(* (jmp_buf *) mem+RBP) != 0) goto labc;`
`FAIL`	`longjmp(* (jmp_buf *) mem+RBP, 1);`
`LAB`	`labc;`
`NOP`	`;`
`CREATELOG`	`log_create(&RLP);`
`REPLAYLOG`	`log_replay(&RLP);`
`REWINDLOG`	`log_rewind(&RLP, b);`
`READ`	`fgets(tmp, sizeof(tmp)-1, stdin); a = atoi(tmp);`
`WRITE`	`printf(b);`
`WRITELN`	`putchar('\n');`

Note that Alma-0 procedures are not implemented as C functions, as this would prevent us from letting the AAA instructions handle the stack. Instead all AAA instructions are translated one-on-one into their C counterparts and placed in one C function. Subroutines are handled by using the `setjmp` mechanism in the standard C library [KR88]. See section 8.4 for a detailed description of the problems encountered here.

The following table provides examples of C code generated by this approach:

`MUL S1, S1, 8`	`RS1 = RS1 * 8;`
`ADD byte[SP+32], S1, 56`	`mem[RSP+32] = RS1 + 56;`
`BEQ word[650], word[646], 43`	`if((* (aaaword_t *) (mem+650)) ==` `    (* (aaaword_t *) (mem+646))) goto lab43;`
`LAB _, _, 43`	`lab43;`
`JSR _, _, 46`	`if(setjmp(* (jmp_buf *) mem+REP) == 0);` `    goto lab46;`
`RTS _, _, _`	`longjmp(* (jmp_buf *) mem+REP, 1);`

8.2 Run-time environment

8.2.1 Memory and registers

For each basic AAA data type an equivalent C type is defined. This type is fixed for all AAA types except for `word`. The actual C type for the AAA type `word` depends on the host computer; it should be large enough to either any values of the Alma-0 type `INTEGER` or a pointer. On most platforms the C type `long` satisfies these conditions.

AAA type	C typedef name	actual C type
byte	aaabyte_t	char
word	aaaword_t	*usually* long
float	aaafloat_t	double
string	aaastring_t	char *

In the AAA run-time environment, the memory of the AAA is represented by an array of type `aaafloat_t` (to get the correct alignment) and the registers are represented by variables of type `aaaword_t`, except for R0 which is defined by a `#define` command as zero, because it should always equal zero. Macro's associate register names with register numbers.

```
aaafloat_t  *mem[AAA_MEM_SIZE / sizeof(aaafloat_t)];
#define     R0      0
aaaword_t   R1, R2, R3, R4, R5, R6;

#define     RZ      0
#define     RS1     1
#define     RS2     2
#define     RLP     3
#define     RBP     4
#define     REP     5
#define     RSP     6
```

8.2.2 Log administration

The log administration system is an important part of the AAA, and its performance has a large impact on the overall performance of the AAA (see section 8.5).

The logs are kept in a singly linked list. The active log is at the front of the list, and the previously active log is its successor.

For every memory block whose value is recorded in the log, a *log entry* is created. The log entries are kept in a binary search tree, as well as in a singly linked list. The binary search tree, which uses the address of the memory block as its key, allows the log administration system to determine quickly whether a memory block starting at the same address has already been recorded in this log. The linked list keeps the log entries in the order they were recorded; new log entries are added to the front of the list. Because traversing a binary tree requires recursion, which may cause stack overflow and impacts performance badly, the linked list is traversed front-to-back instead, when the log is replayed.

Because only the address of a memory block, and not its size, is used as the key for the binary search tree, one memory location is recorded in the log twice, when it is contained by two overlapping memory blocks being recorded. Fortunately, the front-to-back traversal of the singly linked list used when replaying the log, causes its oldest value to be restored last. Therefore, the singly linked list is actually essential to the correct function of the log administration system.

Logs and log entries are represented by the following C types:

```
typedef struct aaalog {
    struct aaalog       *next;
    struct log_entry    *entries;
    struct log_entry    *tree;
} aaalog_t;
```

- The `next` field points to the previously active log.

- The `entries` field points to the most recently added log entry.
- The `tree` field points to the top of the binary tree of log entries.

```
struct log_entry {
    struct log_entry    *next;
    struct log_entry    *lhs, *rhs;
    void                *memaddr;
    aaaword_t           memsize;
    char                memval[1];
};
```

- The `next` field points to the log entry added previously to this one.
- The `lhs` and `rhs` fields point to the left hand and right hand child of this log entry.
- The `memaddr` and `memsize` fields indicate the address and size of the memory block recorded.
- The `memval` field contains the value of the memory block. Although this field is declared to contain only a single byte, a sufficient amount of memory is actually allocated for the `log_entry` record, to save the complete memory block. This little C trick avoids having to allocate two blocks of memory per log entry.

8.3 Generating a valid C program

We have seen how the C statements are generated, and which variable definitions and functions are needed. Now all that remains is generating a valid C program. This is done as follows:

1. The C output file `a.out.c`, which will contain the valid C program, is opened.
2. Dynamic definitions, i.e. the size of main memory, and debugging options, are written to the output file.
3. Type and variable definitions for the run-time system are written.
4. Debugging functions are written.
5. Log administration functions are written.
6. The `main()` function, which initializes the run-time environment and calls the function `aaacode()`, is written.
7. The first part of the function `aaacode()` is written.
8. The C statements corresponding to the AAA code of the main program, followed by the code for the procedures, are written as part of the function `aaacode()`.
9. The closing curly brace for the function `aaacode()` is written.
10. The output file is closed.

The lines written from step 3 to step 7 are actually copied directly from the file `a0cprefix.txt`. This is not a valid file, because it has only a partial definition of the function `aaacode()`. The rest of its definition is provided by steps 8 and 9.

Commentary

8.4c Program counter emulation

The fact, that the C language does not allow the programmer access to the program counter register of the host computer, has serious implications for the implementation of the flow control instructions.

An AAA instruction which jumps to a location known at compile-time, can be translated into a C `goto` statement which jumps to a label which is inserted at the target location by the `LAB` statement.

However, when the target location is not known at compile-time this simple approach does not work. As an example we will take the `JSR` and `RTS` instructions, which are used to call and return from a subroutine. When the `JSR` instruction is executed, the current value of the program counter is saved, and a jump is made to a known location. The corresponding `RTS` instructions jumps back to the original location by restoring the saved value. This location is not known are compile-time, therefore the `RTS` instruction cannot be translated into a `goto` statement.

The `setjmp` mechanism implemented by the standard C library [KR88] provides the solution. The function `setjmp()` saves the current CPU context (program counter and other registers) at a specified address and returns the value zero. When this context is later restored by calling `longjmp()`, execution continues in the function `setjmp()`, which now returns the value given to the function `longjmp()` [1]

When the `JSR` instruction is executed, the function `setjmp()` is called to save the current context at the memory location indicated by the `EP` register (`mem[EP]`). The function `setjmp()` returns 0, and the branch is taken to the subroutine. When the corresponding `RTS` is executed, the function `longjmp()` is called with a value of 1. The original context is restored and the function `setjmp()` returns a second time, this time with the value 1 and thus the branch is not taken.

For the `ONFAIL` en `FAIL` instructions a similar approach is used.

8.5c Alternative log implementations

Before deciding to use the binary search tree implementation, three other implementations of the log administration system were investigated. All implementations keep the logs in a singly linked list, but they differ in their representation of the log:

Linked list A log is represented a singly linked list. When a value is recorded, it, and its memory address, are added to the front of the list. When the log is replayed, the list is traversed front-to-back, and all values encountered are restored. There may be more than one record for the same memory address in the list, but, because the list is replayed front-to-back, eventually the correct value will be restored.

Block copy This implementation does not keep a separate record for each value. Instead when a new log is created, a copy of all memory is made, which is copied back when the log is replayed.

However, This implementation does not function correctly in the presence of `FORALL` statements. The `DO` part of a `FORALL` statement is executed in the context of the surrounding code. Therefore,

[1] Only the adventurous use `longjmp(0)`

during execution of the DO part, the top of the log stack is temporarily made inactive and the log, which was active before the FORALL statement, is temporarily made active to ensure that values assigned in the DO part of the FORALL statement persist when the FORALL statement completes. Because the block copy implementation saves and subsequently restores *all* memory addresses, backtracking in the FORALL part of a FORALL statement causes assignments in the DO part to be undone as well.

Binary tree A log is represented as a regular binary search tree, which uses the memory address as the key. When a value is recorded, it, and its memory address, are added to the tree, provided the same memory address was not added before. If it was, nothing happens.

Apart from being stored in a tree structure, the values are also added to a singly linked list, in the same order as the linked list implementation. Because of this, replaying the log only requires a simple list traversal instead of a recursive tree traversal.

AVL tree This implementation is identical to the binary tree implementation, apart from the fact that the binary search tree has been implemented as an AVL tree [HSAF93].

8.5.1c Performance of the log implementations

In order to determine which implementation gives the best CPU and memory performance, tests were made. A collection of programs (see appendix B) was ran on one same system (a SUN Sparc system with 2 CPU's, 170 MB real memory, 646 MB virtual memory, and SunOS version 5.4 (Solaris)), while measuring the total execution time, the number of allocations requests and the maximum amount of bytes allocated at one time:

Program	knapsack	queens			squares
		N=8	N=9	N=10	
Linked list implementation					
Execution time in seconds	0.800	3.400	16.000	84.090	72.290
# of allocation requests	39282	146742	727981	3745458	1656637
max. # of bytes allocated	2276	3047	3790	4568	11420
Binary tree implementation					
Execution time in seconds	0.710	2.560	13.340	63.020	57.310
# of allocation requests	29918	52348	238745	1137788	629751
max. # of bytes allocated	2609	1638	1854	2070	2721
AVL tree implementation					
Execution time in seconds	0.940	2.900	13.830	69.440	62.170
# of allocation requests	29918	52348	238745	1137788	629751
max. # of bytes allocated	2985	1878	2126	2374	3129
Block copy implementation					
Execution time in seconds	n/a	18.880	85.570	455.290	613.520
# of allocation requests	n/a	13756	64337	313336	474319
max. # of bytes allocated	n/a	289044	321240	353452	323960

We can make the following conclusions from these measurements:

- The binary tree implementation is always quicker than the other implementations, and the maximum number of bytes allocated is nearly always lowest. Therefore the binary tree implementation was chosen as the default implementation[2].

[2] One of the other implementations can be chosen by changing one line in the file a0cprefix.txt

- Although one may expect the linked list implementation to be quicker than the binary tree implementation because very little has to be done to record a value, it is actually slower. It seems that the higher number of allocation requests done offsets any benefits that may be had from the simplicity of the implementation.
- Although the block copy implementation did not function correctly, its performance was still measured, where possible, to determine whether it might be interesting to look for a work-around. However, in some occasions, the block copy implementation is more than ten times slower than the binary tree implementation. The copying seems to incur a great overhead, which may be even greater if the memory size were increased.
- The AVL tree implementation never outperforms the binary tree implementation. One may expect the AVL trees to perform better when they get larger, but, apparently, the trees never get large enough for the smaller height of the AVL trees to offset the performance loss caused by the more complicated implementation.

Conclusions and further work

In this report I have demonstrated how a compiler can be built, which supports the extensions proposed in [AS97]. We have seen how the traditional architectures, which are used to support imperative programming languages, can be extended in a natural way into an architecture, which provides the primitives needed to implement the extensions. We have also seen how the new language constructs can easily be translated into sequences of instructions for this abstract architecture.

Only the implementations of the **SOME**, **KNOWN** and the **MIX** extensions differ somewhat from their original definition, but the spirit of these extensions has been kept intact. This is supported by the fact that the example programs from the original article can be compiled and run (after adding some syntactical sugar to make them into valid Alma-0 programs, see appendix B and the files in the `examples` directory).

We can conclude that it is possible to implement a programming language that supports the proposed extensions. However, there is still work to be done:

- Optimization techniques used by compilers for regular imperative programming languages, could be implemented, e.g. peephole optimization, register allocation strategies, and more constant folding.
- The AAA instructions could be translated into machine language instructions for a certain platform. This would make it possible to compare Alma-0 to C fairly.
- A number of Modula-2 features, which are missing in this version, could be implemented, e.g. modules, more types, more flow control statements.

Furthermore, optimization techniques that concern themselves with the specific features of Alma-0 could be developed. For example, some assignments are never undone, so that the previous value does not need to be recorded. An optimization technique could recognize these assignments and replace the recording version of the instruction with its non-recording version. Another lead is the fact that some variables are assigned a value before their first use, and are therefore always initialized when used in an expression. The flag that signifies whether such a variable is initialized could be removed, as could the instructions testing the flag.

Appendix A

Syntax overview

This appendix gives an overview of the Alma-0 syntax. The rule numbers are also used in the source files `a0gram.l` and `a0gram.y`.

A.1 Lexical rules

The following regular expressions form the rules by which tokens are recognized. The `NEWLINE`, `LAYOUT`, and `COMMENT` tokens are not passed on to the parser, they are used to separate the other tokens from each other. The `NEWLINE` token is also used to keep track of the current line number.

```
1.  NEWLINE = \n
2.  LAYOUT  = [ \t]*
3.  COMMENT = \(\*([^\*]|(\*[^\)]))*\*?\*\)
4.  IDENT   = [a-zA-Z][a-zA-Z0-9]*
5.  INTEGER = [0-9]+
6.  REAL    = [0-9]+\.[0-9]+(E(\+|\-)?[0-9]+)?
7.  STRING  = "[^"]*"  |  '[^']*'
8.  for every keyword there is a rule like:
    FOR     = FOR
```

A.2 Syntax

The syntax is described by a (non-extended) BNF grammar, but take of the following points:

- The symbol $\rightarrow$ has been replaced by the symbol `:`.
- Identifiers that start with a capital are tokens defined by the lexical rules.
- Identifiers that start with a lowercase letter are non-terminals.
- Character sequences encoded in single quotes (e.g. `'='`) are literals.
- The top symbol is `compilationUnit`
- Lines marked with a star (`*`) have been added to implement the extensions.

```
10. number               : INTEGER
                         | REAL
11. identList            : identList ',' IDENT
                         | IDENT
12. qualident            : qualident '.' IDENT
                         | IDENT
13. optQualident         : qualident
                         |      /* EMPTY */
14. designator           : qualident
                         | designator '.' IDENT
                         | designator '[' statementListList ']'       *

15. constDeclaration     : IDENT '=' expression
16. constDeclarationList: constDeclarationList constDeclaration ';'
                         | constDeclaration ';'

17. typeDeclaration      : IDENT '=' type
18. typeDeclarationList : typeDeclarationList typeDeclaration ';'
                         | typeDeclaration ';'
19. type                 : qualident
                         | '(' identList ')'
                         | optQualident '[' expression '..' expression ']'
                         | ARRAY typeList OF type
                         | RECORD fieldListList END
20. typeList             : typeList ',' type
                         | type
21. fieldList            : identList ':' type
                         |      /* EMPTY */
22. fieldListList        : fieldListList ';' fieldList
                         | fieldList

23. varDeclaration       : IdentList ':' type
24. varDeclarationList   : varDeclarationList varDeclaration ';'
                         | varDeclaration ';'

25. statement            : designator ':=' expression
                         | RETURN optExpression
                         | expression                                 *
                         |      /* EMPTY */
26. statementList        : statementList ';' statement
                         | statement
27. statementListList    : statementListList ',' statementList        *
                         | statementList                              *
28. expression           : simpleExpression '=' simpleExpression
                         | simpleExpression '#' simpleExpression
                         | simpleExpression '<>' simpleExpression
                         | simpleExpression '<' simpleExpression
                         | simpleExpression '<=' simpleExpression
                         | simpleExpression '>' simpleExpression
                         | simpleExpression '>=' simpleExpression
                         | simpleExpression
29. optExpression        : expression
                         |      /* EMPTY */
30. simpleExpression     : simpleExpression '+' term
```

```
                        | simpleExpression '-' term
                        | simpleExpression OR term
                        | term
31. term                : term '*' factor
                        | term '/' factor
                        | term DIV factor
                        | term MOD factor
                        | term REM factor
                        | term AND factor
                        | factor
32. factor              : '+' factor
                        | '-' factor
                        | NOT factor
                        | number
                        | STRING
                        | designator
                        | designator '(' statementListList ')'        *
                        | IF statementList THEN statementList         *
                            elsifList optElse END
                        | WHILE statementList DO statementList END    *
                        | REPEAT statementList                        *
                            UNTIL statementList END
                        | FOR IDENT ':=' statementList TO             *
                            statementList optBy DO statementList END
                        | EITHER statementList orelseList END         *
                        | SOME IDENT ':=' statementList TO            *
                            statementList optBy DO statementList END
                        | COMMIT statementList END                    *
                        | FORALL statementList DO statementList END   *
                        | READ '(' statementListList ')'              *
                        | WRITE '(' statementListList ')'             *
                        | WRITELN '(' statementListList ')'           *
                        | WRITELN                                     *
                        | KNOWN '(' designator ')'                    *
                        | '(' statementList ')'                       *
33. elsifList           : ELSIF statementList THEN statementList      *
                            elsifList
                        |       /* EMPTY */
34. optElse             : ELSE statementList
                        |       /* EMPTY */
35. optBy               : BY expression
                        |       /* EMPTY */
36. orelseList          : orelseList ORELSE statementList             *
                        | ORELSE statementList                        *

37. procedureDeclaration: procedureHeading ';' block IDENT
                        | procedureHeading ';' FORWARD
38. procedureHeading    : PROCEDURE IDENT formalParameters
39. block               : declarationList BEGIN statementList END
40. declaration         : CONST constDeclarationList
                        | TYPE typeDeclarationList
                        | VAR varDeclarationList
                        | procedureDeclaration ';'
41. declarationList     : declarationList declaration
```

```
                         |      /* EMPTY */
42. formalParameters    : optInputParamsList optFormalReturnType
43. optInputParamsList  : '(' inputParamsList ')'
| '(' ')'
                         |      /* EMPTY */
44. inputParamsList     : inputParamsList ';' inputParams
                         | inputParams
45. inputParams         : callMech identList ':' type
46. callMech            : VAR
                         | MIX                                          *
                         |      /* EMPTY */
47. optFormalReturnType : ':' qualident
                         |      /* EMPTY */

48. programModule       : MODULE IDENT ';' block IDENT '.'
49. compilationUnit     : IMPLEMENTATION programModule
                         | programModule
```

Appendix B

Example Alma-0 programs

B.1 knapsack.a0

```
MODULE knapsack; (* problem 8 *)
CONST   N = 20;
TYPE    RealVector = ARRAY [1..N] OF REAL;
        BinaryVector = ARRAY [1..N] OF [0..1];

PROCEDURE knapsack(Volume,Value: RealVector; capacity: REAL;
                    VAR Solution: BinaryVector);
    VAR i: INTEGER;
        CurrentBest, TotalValue, volume, waste: REAL;
        CurrentSolution: BinaryVector;
    BEGIN
        CurrentBest := 0.0;
        TotalValue := 0.0;
        FOR i := 1 TO N DO
            TotalValue := TotalValue + Value[i];
        END;
        volume := 0.0;
        waste := 0.0;
        FORALL
            FOR i := 1 TO N DO
                EITHER
                    CurrentSolution[i] := 1;
                    volume := volume + Volume[i];
                    volume <= capacity;
                ORELSE
                    CurrentSolution[i] := 0;
                    waste := waste + Value[i];
                    waste < TotalValue - CurrentBest;
                END
            END
        DO
            CurrentBest := TotalValue - waste;
            Solution := CurrentSolution;
        END;
```

```
    END knapsack;

VAR
    i           : INTEGER;
    capacity    : REAL;
    vols, vals  : RealVector;
    sol         : BinaryVector;
BEGIN
(* initialize problem *)
    vols[1] := 20.0;    vols[2] := 5.0;     vols[3] := 6.3;
    vols[4] := 1.2;     vols[5] := 83.67;   vols[6] := 0.08;
    vols[7] := 3.0E5;   vols[8] := 30.0;    vols[9] := 3.0;
    vols[10] := 16.5;   vols[11] := 19.0;   vols[12] := 10.2;
    vols[13] := 17.4;   vols[14] := 5.0;    vols[15] := 4.0;
    vols[16] := 4.5E2;  vols[17] := 34.0;   vols[18] := 0.43;
    vols[19] := 6.5;    vols[20] := 10.3;

    vals[1] := 19.0;    vals[2] := 10.2;    vals[3] := 17.4;
    vals[4] := 5.0;     vals[5] := 4.0;     vals[6] := 4.5E2;
    vals[7] := 34.0;    vals[8] := 0.43;    vals[9] := 6.5;
    vals[10] := 10.3;   vals[11] := 20.0;   vals[12] := 5.0;
    vals[13] := 6.3;    vals[14] := 1.2;    vals[15] := 83.67;
    vals[16] := 0.08;   vals[17] := 3.0E5;  vals[18] := 30.0;
    vals[19] := 3.0;    vals[20] := 16.5;

    capacity := 50.0;
(* solve problem *)
    knapsack(vols, vals, capacity, sol);
    FOR i := 1 TO N DO
        WRITE('Item ', i);
        IF sol[i] = 0 THEN
            WRITE(' not');
        END;
        WRITELN(' included (volume=', vols[i],
                              ', value=', vals[i], ')');
    END;
END knapsack.
```

B.2 present.a0

```
MODULE present; (* problem 9 *)
CONST   N = 5;
TYPE    reeks = ARRAY [1..N] OF INTEGER;

PROCEDURE find(MIX e: INTEGER; a: reeks);
    VAR i   : INTEGER;
    BEGIN
        SOME i := 1 TO N DO e = a[i] END
    END find;

VAR a, b    : reeks;
    i, x    : INTEGER;
```

```
BEGIN
(* initialize problem *)
    FOR i := 1 TO N DO a[i] := i; b[i] := i END;
    a[3] := 2; b[2] := 6;
(* solve problems *)
    FORALL find(x, a); find(x, b) DO
        WRITELN(x, ' is present in both a and b')
    END;

    WRITE('Enter number to search for: '); READ(i);
    IF find(i, a) THEN WRITELN(i, ' is present in a') END;
    IF find(i, b) THEN WRITELN(i, ' is present in b') END;

    IF FORALL find(x, a) DO find(x, b) END THEN
        WRITELN('All elements of a are present in b');
    ELSE
        WRITELN('Not all elements of a are present in b');
    END;
END present.
```

B.3 queens.a0

```
MODULE queens; (* problem 10 *)
CONST   N = 8;
TYPE    board = ARRAY [1..N] OF [1..N];

PROCEDURE queens(MIX x: board);
    VAR     i, column, row: [1..N];
    BEGIN
        FOR column := 1 TO N DO
            SOME row := 1 TO N DO
                FOR i := 1 TO column-1 DO
                    x[i] <> row;
                    x[i] <> row+column-i;
                    x[i] <> row+i-column
                END;
                x[column] = row
            END
        END
    END queens;

VAR     b       : board;
        nrSols  : INTEGER;
BEGIN
(* solve problem *)
    nrSols := 0;
    FORALL  queens(b);
    DO      nrSols := nrSols + 1;
    END;
    WRITELN('Number of solutions = ', nrSols);
END queens.
```

B.4 squares.a0

```
MODULE squares; (* problem 11 *)
CONST   NX = 5; NY = 6;
        M = 10;
TYPE    SquaresVector = ARRAY [1..M] OF INTEGER;

PROCEDURE AlreadyCovered(i, j: INTEGER;
                        Sizes: SquaresVector;
                        MIX PosX, PosY: SquaresVector);
    VAR h   : INTEGER;
    BEGIN
        SOME h := 1 TO M DO
            KNOWN(PosX[h]);
            KNOWN(PosY[h]);
            PosX[h] <= i;
            PosX[h] + Sizes[h] > i;
            PosY[h] <= j;
            PosY[h] + Sizes[h] > j;
        END
    END AlreadyCovered;

PROCEDURE Squares(Sizes: SquaresVector;
                MIX PosX, PosY: SquaresVector);
    VAR i, j, k : INTEGER;
    BEGIN
        FOR i := 1 TO NX DO
            FOR j := 1 TO NY DO
                IF NOT AlreadyCovered(i,j,Sizes,PosX,PosY) THEN
                    SOME k := 1 TO M DO
                        Sizes[k] + i <= NX + 1;
                        Sizes[k] + j <= NY + 1;
                        PosX[k] = i;
                        PosY[k] = j;
                    END
                END
            END
        END
    END Squares;

VAR
    i                   : INTEGER;
    sqSizes, sqXs, sqYs : SquaresVector;
BEGIN
(* initialize problem *)
    sqSizes[1] := 1; sqSizes[2] := 2; sqSizes[3] := 2;
    sqSizes[4] := 1; sqSizes[5] := 2; sqSizes[6] := 1;
    sqSizes[7] := 3; sqSizes[8] := 2; sqSizes[9] := 1;
    sqSizes[10] := 1;
(* solve problem *)
    IF Squares(sqSizes, sqXs, sqYs) THEN
        FOR i := 1 TO M DO
            WRITELN('Square ', i, ' of size ', sqSizes[i],
```

```
                        ' at ', sqXs[i], ',', sqYs[i])
            END
        ELSE WRITELN('No solution');
        END;
END squares.
```

Bibliography

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991.

[AS97] Krzysztof R. Apt and A. Schaerf. Search and imperative programming. In *Proc. 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 67–79, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[DS95] Charles Donnoly and Richard Stallman. *Bison, the YACC-compatible Parser Generator*. Free Software Foundation, Cambridge, Massachusetts, May 1995. Available online at `http://www.math.utah.edu/docs/info/bison_toc.html`.

[ES90] Margaret E. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, Massachusetts, 1990.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, Version 1.0*. Sun Microsystems, August 1996. Available online at `http://java.sun.com/docs/language_specification/index.html`.

[HSAF93] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of Data Structures in C*. W.H. Freeman and Company, New York, 1993.

[Joh78] S. C. Johnson. YACC: Yet Another Compiler-Compiler. Technical report, Bell Laboratories, Murray Hill, 1978.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall, Eaglewood Cliffs, New Jersey, second edition, 1988.

[Les75] M.E. Lesk. Lex, a lexical analyzer generator. Technical report, Bell Laboratories, Murray Hill, 1975.

[Pax95] Vern Paxson. *Flex, version 2.5, A fast scanner generator*. The Regents of the University of California, March 1995. Available online at `http://www.math.utah.edu/docs/info/flex_toc.html`.

[Wir76] Niklaus Wirth. *Algorithms + data structures = programs*. Prentice Hall, Eaglewood Cliffs, New Jersey, 1976.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York, third, corr. edition, 1985.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison Wesley, Reading, Massachusetts, 1996.