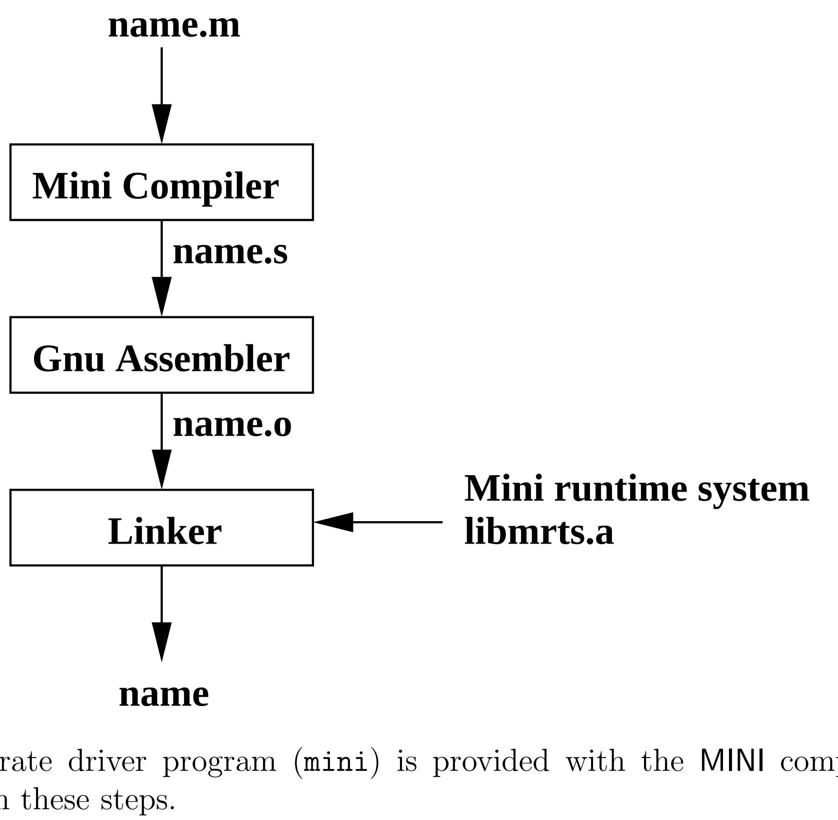
Description of the MINI Compiler and the MINI RunTime System

M. Seutter

September 11, 2004

1 Overview

The MINI compiler (`mini-compile`) translates programs written in MINI to equivalent programs written in Assembler for an Intel IA-32 (ix86) or a Sparc architecture. Such an Assembler program is then translated into an object file and linked with the MINI runtime system to create an executable:



A seperate driver program (`mini`) is provided with the MINI compiler to perform these steps.

2 Installation

After gunzipping and untarring the file `mini-1.2.tgz`, you should have one directory called `mini` with the following files and subdirectories in it:

- `Makefile` Makefile to make the runtime library, MINI compiler and driver.
- `doc` Directory containing the source files of the documentation.
- `gen` Directory containing the source files of the MINI compiler.
- `mrts` Directory containing the source files of the MINI runtime system.
- `driver` Directory containing the source files of the driver.
- `scripts` Directory containing the exercise handin script.
- `ex1` Directory containing some test files for the first exercise.

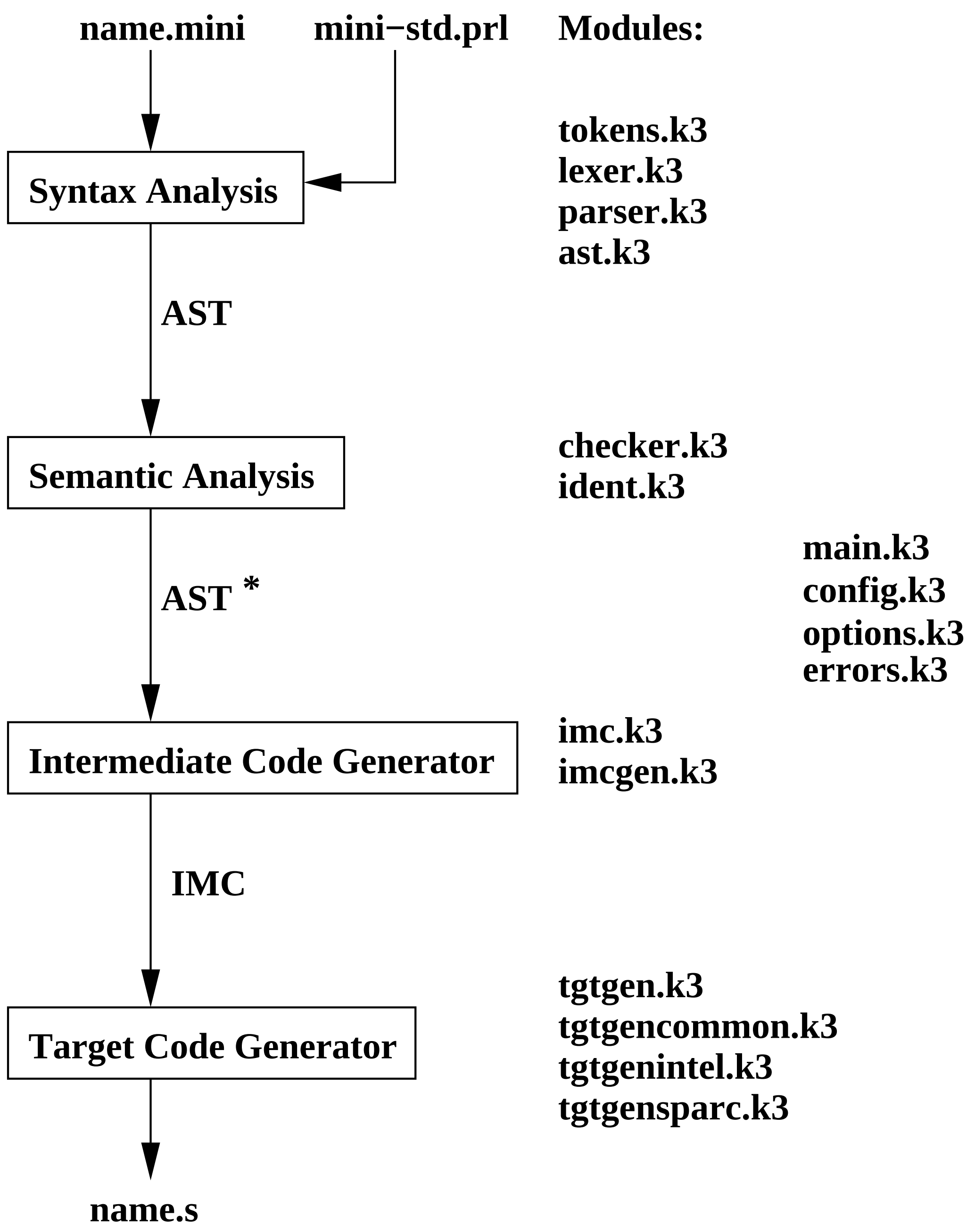
Next, adapt the top `Makefile` to suit your system. To ease the handing in of the compiler construction exercises, the indicated settings are strongly recommended. Secondly, adapt the `Makefile` in the `gen` subdirectory to set the location at which you have installed the CDL3 compiler. Then enter `make`. After completion, you should have 4 extra subdirectories:

- `bin` Directory containing the compiler and driver executables `mini-compile` and `mini`, and the exercise handin script `mkhandin`.
- `lib` Directory containing the runtime system `libmrts.a`.
- `include` Directory containing the runtime system header files.
- `etc` Directory containing the file `mini-std.prl` which declares all predefined operators and procedures.

3 The MINI compiler

3.1 Basic structure of the MINI compiler

The MINI compiler itself is written in CDL3. It translates MINI programs in 4 passes, which communicate through common datastructures:



The first pass does lexical and syntactical analysis. The lexical analysis recognizes program symbols, which are read by the syntactical analysis. The syntactical analysis recognizes the structure of the program and constructs the so-called abstract syntax tree (AST), representing this structure. Before reading a MINI program, the syntactical phase reads a file called `mini-std.prl`, which defines all predefined operations contained in the MINI compiler and runtime environment. These definitions are incorporated in the abstract syntax tree, which is forwarded to the second pass of the compiler.

The second pass of the compiler is responsible for semantical analysis. During semantical analysis, objects, procedures and operators are identified and context conditions are checked. While the compiler is checking, it decorates the abstract syntax tree with extra information, which eases the next pass of the compiler.

The third pass of the compiler translates program constructs into machine independent, yet close to actual hardware, abstract machine code. This intermediate code (IMC) is based on quadruple notation and is especially suited for machine independent optimization. However, during this course we will not study optimization.

Finally, the fourth pass of the compiler translates the intermediate code into Assembler for the target architecture. This translation is quite straightforward. Typically, one instruction in the intermediate code corresponds to two or three instructions in Assembler. Since no register optimization is performed, the resulting code leaves much to be desired in efficiency.

The previous figure also shows the module decomposition of the MINI compiler.

3.2 Common modules

Four modules are common to all modules of the MINI compiler:

`main.k3` This module defines the main program of the MINI compiler.

`config.k3` This module defines the default configuration of the MINI compiler and is *generated* when the MINI compiler is built.

`options.k3` This module parses the command line for compiler options and the name of the source file.

`errors.k3` This module is responsible for error handling and debugging output.

3.3 Syntactical phase

Four modules comprise the first pass of the compiler:

`tokens.k3` This module describes the structure of the tokens output by the lexer and read by the parser. It also provides some basic conversion procedures.

`lexer.k3` The lexer is responsible for the lower level reading of both the standard library (`mini-std.prl`) and the MINI source file. Layout and comments are discarded by the lexer, while tokenizing the input.

`ast.k3` This module describes the structure of the abstract syntax tree and provides some basic conversion procedures for parts of this syntax tree.

`parser.k3` The parser recognizes the structure of the source program and the structure of the declarations in the standard library and builds a (combined) abstract syntax tree for these constructs. It is the largest (and most documented) module of the MINI compiler.

3.4 Semantical phase

The second pass of the compiler consists only of two modules:

`checker.k3` This module checks if the source file (and the standard library) conform to the context conditions described in the MINI language description by traversing and decorating the abstract syntax tree (or more accurately by breaking down the abstract syntax tree and reconstructing a decorated abstract syntax tree from the parts).

`ident.k3` This module checks prior to the traversal of the abstract syntax tree, whether it contains any (conflicting) double declarations. During the traversal, it tries to find for every application of a name a corresponding definition. Since MINI allows generic (overloaded) procedure and operator declarations, both tasks are not trivial.

3.5 Intermediate code generation

Two modules comprise the third pass of the compiler:

`imc.k3` This module describes the structure of the intermediate code (IMC) and some basic conversion procedures. One may view this intermediate code as a linear array of symbolic instructions, some of which carry labels to allow them to be used as targets of jumps.

`imcgen.k3` This module translates the source program into an equivalent program in the intermediate code by traversing the abstract syntax tree while coding the program.

3.6 Target code generation

The last phase of the compiler consists of four modules:

`tgtgen.k3` This module does some initialization for code generation and then calls the correct target code generator as determined from the options and compiler configuration.

`tgtgencommon.k3` This module contains all procedures that are common to the two target code generators.

`tgtgenintel.k3` This module translates the intermediate code into Intel IA-32 Assembler.

`tgtgensparc.k3` This module translates the intermediate code into Sparc (< V9) Assembler.

4 Runtime model

4.1 Memory model and object representation

The runtime model for MINI is based on the usual C conventions of the target machines (with some additions). We assume that memory consists of words that are four bytes long and aligned to four byte boundaries (Note that this excludes the V9 Sparc architecture). Addresses however are byte addresses.

The memory is partitioned in three parts, called the text, data and stack segment, thus conforming to the usual C and Unix conventions. The text segment holds the program code. The data segment holds `TEXT` denotations and the heap. This heap grows towards higher addresses. The stack is used for procedure return addresses, actual parameters (when required by the C convention for the target machine (Note Sparc)), local variables and intermediate results and grows from high to low addresses.

Integers, booleans and addresses are represented by their actual values. A `TEXT` is represented by a pointer, pointing to a null terminated string of characters, residing in the heap (when dynamically allocated) or in the data segment (when pointing to a `TEXT` denotation). The null pointer is used to represent *uninitialized* `TEXT` values. Thus all objects in MINI use one machine word on our target architectures (and can therefore be loaded in a machine register).

4.2 Incremental garbage collection

To support reuse of memory, an *incremental garbage collector* is implemented for all objects in the heap. All objects in the heap are therefore preceded by two *short* integers. The first of these shorts denotes the number of pointers pointing to the object, the so-called *reference count*. The other object denotes the *actual* object length. To allow ease of reuse, all objects are allocated on word aligned addresses. Objects may therefore be padded with bytes, thus causing the actual object length to be larger than the needed object length.

The basic idea of incremental garbage collection is that the reference count of the object is incremented when a pointer starts to point to the object and decremented when a pointer no longer points to the object. When the reference count of an object drops to 0, it has become unreachable from the running program and can therefore be reused. For historic reasons, these two operations are called *attach* and *detach*. The `TEXT` assignment in MINI can then be described by the following three simple steps:

Attach the source object
Detach the destination object
Copy source to destination (Only an address)

The reader may figure out for himself why the attach operation *must* be executed first.

`TEXT` denotations have the same layout as heap objects. They differ however in two aspects. They are allocated in the constant part of the data segment. To indicate that they are constant and should never be recycled by the garbage collector, their reference count equals -1.

The following three functions are available to interact with the garbage collector:

`char *rts_malloc (int size)`
: This function allocates an object able to hold `size` bytes either from the internal free list administration or from the heap. The initial reference count of this object is (of course) 1. It is guaranteed that the allocated object is word aligned.

`void rts_attach (char *ptr)`
: This function increases the reference count of the object, `ptr` is pointing to. When `ptr` is a null pointer, `rts_attach` aborts. When the original reference count was -1, `rts_attach` returns directly.

`void rts_detach (char **ptr)`
: This function decreases the reference count of the object, `*ptr` is pointing to, while writing null to `*ptr`. When the reference count becomes

0, the object is moved to the internal free list administration. When `*ptr` is a null pointer or is pointing to a `TEXT` denotation, `rts_detach` returns directly.

4.3 The runtime library `libmrts.a`

The runtime library currently consists of the following five modules:

`rts_error` This module gives runtime errors.

`rts_alloc` The garbage collector. See above.

`rts_ints` This module holds support routines for `INT` operations.

`rts_texts`
: This module holds support routines for `TEXT` operations. In exercise 3 (or 4) we will extend this set of operations.

`rts_xput`
: This module is responsible for reading and printing of objects.

In the directory holding the source code of these modules, a small test program is also present to test the functionality of the library.