

Engineering project Compiler Construction

Assignments

August 31, 2009

1 Introduction

This engineering project on compiler construction consists of three assignments. We start from an existing compiler for a simple and extensible language, called Asterix. The compiler is written in ANSI-C, and also generates ANSI-C. The compiler has to be modified and extended, as is described in the following section. The main purposes of this course are to get acquainted with modern compiler writing tools, and to develop algorithms used for compiling contemporary language paradigms.

We strongly suggest that the assignments be implemented by groups of two persons, because of the amount of work involved. It is not allowed to work with groups of more than two persons. When working as a team you are *not* allowed to distribute the work and have each assignment being completed by one student, because that defeats the objective of obtaining detailed hands-on experience with all three selected topics.

Carefully read this document before starting programming.

2 Assignments

All three assignments are mandatory. The first is on parsing, the second on classes, objects, and inheritance, and the third on garbage collection.

2.1 Assignment 1

The first assignment is on tools for lexical analysis and parsing. The Asterix compiler from which we start, is implemented using the Flex (Lex-like) lexical analyzer generator and the Bison (YACC-like) LALR parser generator. Once they were very innovative tools, but nowadays there are much more modern parser generating tools, which are easier to use, faster, and much better at error correction.

In this assignment the Bison implementation has to be replaced by an LLnextgen implementation. LLnextgen is a rewrite of LLgen, hence the name, and provides better development support (error reporting) than the original LLgen implementation, which is a modern LL(1) parser generator, with static and dynamic facilities to resolve non-LL(1) ambiguities. The behavior of the new compiler must be the same as the old compiler for correct input. For incorrect input, the compiler with the LLnextgen parser must have better support for error reporting and error correction.

The fact that the original parser generator is LALR and the new one is LL(1) has a number of implications:

- Rewrite the grammar rules in such a way that they are LL(1).

- Use the constructs for repetition ("*" and "+") if this is more natural than recursion. In Bison recursion is the *only* way to implement repetition.
- Use the construct for optional sub-rules ("?") instead of the verbose way this done in Bison.
- With a Bison parser, parsing stops as soon as a parse error is encountered. With the LL-nextgen parser, the compiler should give a sane error message, such as "file.src, 22: expected <expression>", continue parsing, and do semantic analysis. Make use of the fact that LL-nextgen will always complete a rule once it has started one. Your compiler will be thoroughly tested with correct input, as well as with erroneous input.

The sources for the Asterix compiler can be found in `/usr/local/asterix-compiler`. You should create your own source directory and copy all files in `/usr/local/asterix-compiler` to that directory.

Tips:

- First rewrite the grammar, temporarily turn off semantic analysis and code generation, and test the parser with correct and incorrect input. If this works as it should work, rewrite the semantic actions.
- Whenever something goes wrong, return a null pointer: for example, return `(Expression *) 0` for a missing expression. The semantic analyzer only assumes tokens to be non-null (see the README file).
- Test your compiler thoroughly; you need it to complete assignment 2.

2.2 Assignment 2

[Grading assignment 1 may take some time, so do not wait for the result and continue with this assignment immediately. However, we do expect you to fix the bugs, if any, discovered by your instructor during the grading of assignment 1, so do make a snap shot of your compiler now.]

In this assignment, you are to extend the Asterix language with objects and classes, inheritance, and dynamic binding. A complete description of the changes of the language specification is given below. You must modify your LLnextgen-based compiler such that it implements the extended language.

`declaration` $\rightarrow$ `class_declaration`

In addition to var declarations and subprogram declarations, we introduce class declarations.

`class_declaration` $\rightarrow$ `"class" identifier1 [ "inherits" identifier2 ] "is" declaration * "end"`

This rule declares a new class with the name `identifier1` in the global scope. It is an error to declare a class other than at the global level. `identifier1` may not be re-declared in the global scope.

Optionally the class may inherit from one superclass `identifier2`. `identifier2` must be declared as a class before `identifier1`.

Each class introduces a new scope for its member declarations. These can be either var declarations or subprogram declarations (methods). If the class does not inherit from another class, the immediate parent scope of the member scope is the global scope. If the class does inherit from another class, the immediate parent scope of the class is the member scope of the class from which it inherits (see Fig. 1).

A method in a superclass can be overridden by a method in a subclass with the same name. Method overriding is only allowed if the types of the arguments are exactly the same, including the way arguments are passed (by value or by reference). Furthermore, both methods must either be a function returning the same type, or both be a procedure. Member variables cannot be overridden, however they can be redeclared in a subclass as a different variable.

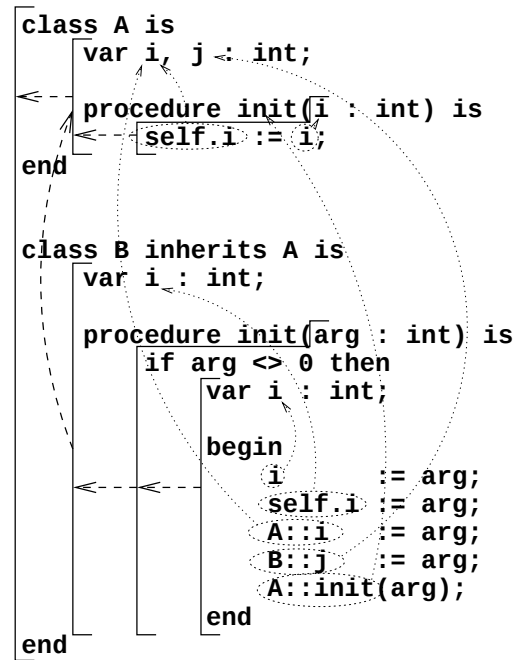


Figure 1: Scopes with classes.

type $\rightarrow$ identifier

A new class type is introduced. **identifier** must be declared by a preceding, or the current, class declaration. A variable of this type is a reference to an object of (a subclass of) class **identifier**. Object references must be initialized to null.

There is a distinction between a static type and a dynamic type. The static type of a variable is the type used in its declaration. The dynamic type of a variable is the type used in the "new" expression.

There is also a distinction between assignment compatibility and being of exactly the same types. Two expressions are assignment compatible if the static type of the lvalue is (a superclass (recursively)) of the static type of the rvalue. Assignment compatibility is used for the assignment statement, call-by-value parameter passing, and function value returning. For call-by-reference parameter passing, the types must be exactly the same (as well as for arrays of objects.)

expression $\rightarrow$ "new" identifier

This expression creates an object of static and dynamic type **identifier**. **identifier** must be declared by a class declaration. If there is no heap space for a new object, a runtime error occurs. The newly created object should be properly initialized, with all member variables set to their default initialization value.

statement $\rightarrow$ "delete" expression ";"

The semantics of a delete-statement has been redefined as deleting a class object or an array object. The reference must have been obtained by a new expression, although deleting a "null" value is allowed. Runtime behavior is undefined if an object is deleted more than once.

expression $\rightarrow$ "self"

"self" is a reference to the object currently being invoked. It may not be used outside a method. Its static type is a reference to the class that contains the method. Its dynamic type is determined at object creation time. It is not an lvalue.

expression → "null"

The meaning of the "null" expression has been redefined. "null" denotes a null reference to either an array of any type or any class. It is not an lvalue.

expression → expression₁ "." qualident

qualident → [identifier₁ "::"] identifier₂

The dot-operator is used to select a member of an object. The precedence level and associativity are the same as for the index operator "[]" and function call "()". expression₁ must be of any class type. The program behavior is undefined if expression₁ equals "null" or has been deleted.

If identifier₁ is specified, it must be the same class as the static type of expression₁, or one of its superclasses (recursively). The qualification operator "::" binds stronger than the dot-operator. Name resolution for identifier₂ starts in identifier₁, or in the static type of expression₁ if identifier₁ is not specified. If identifier₂ is not declared in that class, and the class has a superclass, identifier₂ is searched for in its superclass (recursively). It is a compile-time error if identifier₂ is not declared in any of its (super)classes.

Binding is dynamic if identifier₂ denotes a method and is not qualified with identifier₁. If identifier₂ denotes a member variable, or is qualified with identifier₁, binding is static. The difference between static and dynamic binding has consequences for the method being invoked. If B is a subclass of A and overrides method m, object obj has static type A and dynamic type B, then if obj.m() is specified, the method defined in class B is invoked; if obj.A::m() is specified, the method defined in class A is invoked.

For convenience the following short-hand notation may be used within the context of a method:

expression → qualident

it is semantically equivalent to "self"."qualident.

An example program is shown below.

```
(* Test function overriding *)

include io.inc

class A is
    function f() : int is return 1;
end

class B inherits A is
    function f() : int is return 2;
end

class C inherits B is
    function f() : int is return 3;
end

function main(argv : array of string) : int is

var obj : B;

begin
    obj := new C;
    WriteString("Expect 3 : ");
```

```

    WriteInt(obj.f());          (* Note: obj.C::f() is not allowed *)
    WriteLine();
    WriteString("Expect 2 : ");
    WriteInt(obj.B::f());
    WriteLine();
    WriteString("Expect 1 : ");
    WriteInt(obj.A::f());
    WriteLine();
    delete obj;

    return 0;
end

```

3 Assignment 3

In this assignment you are to implement compiler and runtime support for automatic garbage collection of arrays and objects. The runtime support part consists of a mark and scan garbage collector. The compiler part consists of generating code that assists the garbage collector in finding all references inside an object.

3.1 Runtime support

You must modify your compiler to generate code that uses a memory allocator that knows about garbage collection. In addition, you must implement a garbage collection routine. The allocator and the garbage collection routine are described below.

The memory allocator

You must modify the runtime system to invoke memory allocation routine `gc_alloc` supplied in the file `gc.c`. `Gc_alloc` allocates memory by finding the first chunk in the free list that is large enough to accomodate the request and splitting off a chunk of the requested size, or the chunk itself if the remainder of the chunk would be of unusable size. Immediately before each chunk of data that is allocated, the allocator stores a header that contains three fields: a size field, a counter field and a marked bit. All fields are needed by the garbage collector that you will implement (see below). The free list is implemented as a singly linked list, using the first bytes of the data part of the chunk as the pointer to the next object.

`Gc_alloc` invokes a garbage collection routine (`gc_collect`) whenever it runs out of memory. Your implementation of this routine must use the mark and scan algorithm described in "Modern Compiler Construction," using the Schorr and Waite pointer reversal technique. Recall that this algorithm performs a depth-first traversal of all objects reachable through the so-called *root set*. During this traversal, all live objects are marked. Then a scan of the heap will add all unmarked blocks to the free list, coalescing adjacent free chunks and resetting the marked bits on all chunks. You must adapt the algorithm to work with Asterix objects and arrays, as described below.

Garbage collection

Whenever the allocator runs out of memory, it will invoke `gc_collect`. `Gc_collect` must mark all live data objects (including arrays). Marking a data object is done by the function `gc_process_pointer_address` (which you must also implement). This function takes the *address* of an array or object reference and marks any arrays or objects reachable from the reference using the Schorr and Waite

pointer reversal technique. Recall that an object may be reachable through multiple references, so `gc_process_pointer_address` must be careful not to process the same object twice. To locate the non-root objects to mark, the garbage collector invokes the objects *scan method* (described in Section 3.2). An object's scan method knows how to find all references inside an object.

After the marking phase, the garbage collector must scan the heap for unmarked (and therefore unused) chunks. As each chunk contains the size of the chunk in its header, the address of the next chunk can be calculated. Starting from the bottom of the heap, all chunks can thus be visited. It is important to also reset the marked bit on chunks that are in use, to ensure that the next `gc_collect` run will work properly. If there are multiple consecutive unused chunks, they must be coalesced into a single unused chunk.

3.2 Compiler support

The garbage collection process described in Section 3.1 depends on compiler support in two ways:

1. The garbage collector needs to know the root set. Via the roots the garbage collector can find every live data item.
2. The garbage collector needs to be able to find references inside objects. The necessary type information is available in the compiler, which knows the structure of every class.

Tracking roots

All live data is accessible via the so-called *root set*. The root set consists of all global references and local references in active subprograms. Your compiler must generate code that keeps track of the root set. The easiest way to do this is to push the *addresses* of all of a subprogram's references onto a global stack when the subprogram is entered. The compiler must also generate code that removes references from the stack when the subprogram exits. Note that references stored in global variables should always be on the stack.

Expressions may allocate more than one new object. This can cause problems if a garbage collection is triggered after a new object has been created. Because the new object's reference has not been assigned to a variable reachable from the root set, it will be considered a free block. However, the block is still necessary for the correct calculation of the expression. This problem can be solved by introducing temporary variables that are added to the root set. Note that the assignment of the object reference to the temporary variable can be done as part of evaluation of the expression, because in C an assignment is also an expression that evaluates to the value being assigned. For example, the code `(tmp = new_array(10, 4))` can be used where previously only the call to `new_array` would appear. Be aware that new objects can also be returned from a function call.

Finding object references

Modify your compiler to generate a *scan method* for each class declared in the source program. Scan methods are used to locate references inside an object. Each scan method has the following signature:

```
object *gc_scan(object ref, int nr);
```

where `ref` is a reference to an object or an array, and `nr` is the number of the child reference to be located. A `gc_scan` method will return the address of the child reference, or `NULL` if `nr` is greater than the number of child references in the object. By returning the address of the child reference,

the location to store the parent pointer in the Schorr and Waite pointer reversal technique is retrieved.

The scan method must be stored in the first entry of the class's dispatch table. This way, your garbage collector can always find the scan method. To simplify calling the first entry of the class's dispatch table, you should modify the definition of `object` as follows:

```
typedef struct base_disp_table base_disp_table;

typedef struct
{
    base_disp_table *disp_table;
    /* other fields here */
} *object;

struct base_disp_table {
    object *(*gc_scan)(object, int);
};
```

Arrays are a bit special. The layout of an array must be changed such that it looks like an object to the garbage collector. This means that the first field of each array structure must point to a "dispatch table", just like objects. This field is then followed by the array's size (in elements), and its data. Note that this layout means an extra level of indirection, but this way the garbage collector can process objects and arrays in the same way, simplifying the implementation.

3.3 Hints

- Start by implementing your garbage collector with a separate pointer in the chunk header for storing the parent pointer. The complex pointer shuffling required for the Schorr and Waite pointer reversal technique can be implemented later, after the rest of the garbage collector has been debugged.
- Test your garbage collector with small heaps. The heap size can be set using the `GC_HEAPSIZE` environment variable.
- Test your garbage collector with objects that are referenced multiple times.
- Don't forget the arguments passed to the `main` function.

4 General remarks

4.1 Deadlines

The assignments should be turned in no later than 08:59 on the dates listed below:

Assignment	Date
1	March 9 th , 2009
2	May 6 th , 2009
3	June 8 th , 2009

For every day that you hand in your assignment late we will subtract 1 point.

4.2 Documentation

The assignments should be accompanied by separate documentation, preferably in PDF format. The documentation should reflect on how you handled the assignment; report on your approach and link it to the general theory of compiler construction as provided by the book and lectures. For instance, for assignment 1 you could indicate where you applied left factoring to make the original Asterix grammar LL(1) and provide an example grammar rule. It is important that you are explicit about the particular problems that you encountered; you must clearly describe them, your solution, used algorithms, and so on. Nevertheless, the documentation must be to the point and concise, so limit your writing to around 4 pages. The documentation will also be graded.

4.3 Instructors

At the beginning of this practical course, you will receive mail that will give you information about who will be your instructor. He will grade your solutions and provide feedback.

Person	Email address	Room
Gertjan Halkes	G.P.Halkes@ewi.tudelft.nl	HB 09.080

An instructor will be present at the labs at Drebbelweg to answer questions and assist with practical problems (e.g., debugging code). Take advantage of this service!

Day	Time	Rooms
Thursday (Feb 5 - March 19)	13:45 - 17:30	DW_0-010-01 ... DW_0-010-04
Wednesday (Apr 8 - Jun 3)	13:45 - 17:30	DW_1-160

For questions about assignment 2 (object-oriented Asterix) you can also request the on-line assistance of Panoramix. This Java applet takes an Asterix program and returns example C-code (cb.h & cb.c) produced by an extended version of the reference compiler. Thus, Panoramix will provide you with a flavor of the target code that your compiler could generate. In addition, you can consult Panoramix to resolve language semantics (“what should dynamic binding do in this test case?”). Enjoy.

4.4 Grading

Grading is done per assignment. According to the following table, points can be obtained for *test results*, *documentation*, and *source code*.

Assignment	Test results	Documentation	Source code
1	16	7	7
2	26	7	7
3	16	7	7

4.5 How to submit your work

- In your own source directory, type
make submit

which will pack the complete compiler into a file named *submit.tgz*. Warning: if you not only modified files, but also added new ones, be sure to add them at the appropriate place in the Makefile. Note: you may **not** modify the cbc script, because our grading tools depend on your compiler behaving “nicely”.

- Test if you can unpack and compile your compiler with


```
tar xzf submit.tgz
make
```

 (Go to some temporary directory before unpacking the compiler.)
- Upload the *submit.tgz* file and your documentation (PDF file) to the CPM system:
 - login to CPM (<http://cpm.ewi.tudelft.nl/>) using your NetID and password from Blackboard.
 - select the 'compiler construction' course.
 - click on the 'right' entry in the overview table, which will pop-up a browse menu for submitting a deliverable; you need to do this twice (sources + documentation).
 - click on the 'notification' button if you want to receive e-mails about the progress of your submission (conformity test, grading, etc.).
 - if your submission does not pass the automatic conformity test by the CPM system (run every 10 minutes), login to CPM again and check the error report, fix the problem, and resubmit. We will not consider your work submitted until the status is set to ScriptApproved.

Do not submit unless you managed to unpack and compile your own sources.

It is useless to turn in your solution far in advance of a deadline; your compiler will be tested and examined (shortly) after the deadline.

4.6 Important announcements

Frequently check the blackboard (in4305); announcements and updates to this course will be posted there.

4.7 Useful hints

- Carefully read the specifications of the language (extensions) and make sure you understand every sentence.
- Stick to the original programming style. Write cleanly, since your code will be thoroughly examined.
- Before turning in your compiler, test it carefully, both with correct and incorrect programs. Generate correct output for correct input, and give clear errors for incorrect input. Make sure your program does not dump core, even on incorrect input (although the use of `assert()` is encouraged).
- Both the compiler and the generated code are ANSI-C. This implies in particular that:
 - you should not generate zero-sized structs;
 - you should not generate zero-sized arrays, or arrays of variable size (use `malloc()` instead);
 - a block may not end with a label (e.g.: write `default::}` instead of `default:}`);
 - `void *` is not compatible with a function pointer; cast to the appropriate type;
 - the last field of an `enum` may not be followed by a comma;
 - you should only use standard library functions that are described by the ANSI-C reference manual;

- you should not rely on the evaluation order of actual arguments.

To test whether the code is ANSI-C conformant, use `gcc -ansi -pedantic`.

A The Asterix Compiler

This appendix provides a short introduction to the source code of the Asterix compiler. Figure 2 shows the structure of the compiler, which is in part generated by tools (flex and yacc), and the transformation process from Asterix source to executable. The $\oplus$ operator denotes compilation and linking by the C compiler (gcc).

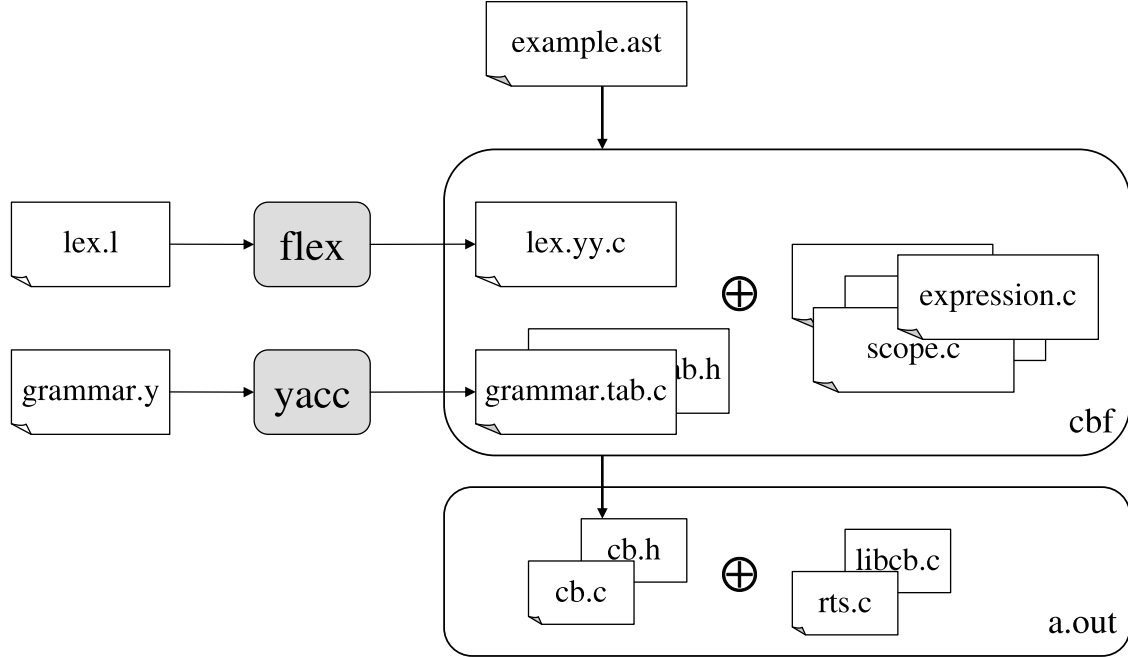


Figure 2: Compiler structure.

A.1 Overview

The design of the compiler is straightforward. It has a lexical analysis, parsing, semantic analysis and code generation phase. The `main()` function (located in `main.c`), activates these phases in sequence.

```
int main(int argc, char **argv)
{
    int yyparse(void);

    init(argc, argv);
    yyparse();

    semantic_analyses();

    if (!error_seen)
        code_generation();

    terminate();

    return error_seen ? 1 : 0;
}
```

Since the lexical analysis code is called by the parser, the first relevant call in `main()` is the one to `yyparse()`, the parser generated by Bison. The result of the semantic actions during parsing is an AST, containing a list of global declarations, which is stored in the variable `global_declarations`. Next, `semantic_analyses()` (invoking `type_check_declaration_list(global_declarations)`) and `code_generation()` (invoking `generate_declaration_list(global_declarations)`) are called to perform the semantic analyses and code generation, respectively. These functions start a traversal of the AST by recursively calling other functions to process the branches. The code for the `type_check_` and `generate_` functions is stored in a number of `.c` files, for instance, `declaration.c` for everything to do with declarations. These files also contain functions to create and delete the structs that make up the AST.

A.2 Flex and Bison

For the first assignment it is important to understand the way Flex and Bison depend on each other. This is summarized in the following table.

Tool	Source	Generates	Depends on
Flex	lex.l	lex.yy.c containing <code>yylex()</code> and <code>yytext</code>	<code>yyval</code> variable token <code>#define</code> 's
Bison	grammar.y	grammar.tab.c containing <code>yyparse()</code> and <code>yyval</code> grammar.tab.h containing token <code>#define</code> 's	<code>yylex()</code>

The code generated by Bison (`yyparse()`) calls the `yylex()` function, which is generated by Flex. Flex also creates the `yytext` variable, containing the string representation of the last recognized token. In a bottom up parser like Bison, all tokens in a rule are consumed before the rule is reduced. Since we may need the actual text of the token (or the position in the source file), Bison creates the variable `yyval`, in which the C code in the lex file can store this information. The `new_token()` function in `lex.l` allocates a structure containing the line number and text of the token. Bison then stores this in the `$n` parameters, which can be used in the Bison code. In a top down parser, tokens are consumed one by one as we process a rule, so we do not need this mechanism. We can just use `new_token()` where we need and `yytext` will contain the last recognized token. Apart from this, the picture when we replace Bison with LLnextgen is similar:

Tool	Source	Generates	Depends on
Flex	lex.l	lex.yy.c containing <code>yylex()</code> and <code>yytext</code>	token <code>#define</code> 's
LLnextgen	grammar.g	grammar.c containing <code>yyparse()</code> grammar.h containing token <code>#define</code> 's	<code>yylex()</code>

Note that the names of the generated files are different.

A.3 Makefile

A Makefile is available in the source directory to build the compiler. It can be used to build the compiler either using Bison or using LLnextgen. Which tool is used can be selected by commenting the appropriate line at the top of the Makefile. To select LLnextgen:

```
#VERSION = Bison
VERSION = LLnextgen
```

Note that the Makefile assumes the grammar file for LLnextgen to be called `grammar.g`.

A.4 Run time system

Building the compiler produces an executable called `cbf`. With this we can compile an Asterix source file:

```
cbf -I<include directory> <asterix source>
```

This results in two files containing C code: `cb.c` and `cb.h`. We can compile these with a normal C compiler like `gcc` to create an executable file. The compiler also has a run time system, containing code on which `cb.c` depends. The runtime system consists of the files `rts.*` and `libcb.*`. We need to link these files when compiling `cb.c`. For convenience a script called `cbc` is available that performs both steps. It compiles an Asterix source file into `a.out`.

```
cbc <asterix source>
```

A.5 Building the syntax tree

The `.c` files contain `new_` and `delete_` functions for every type of node in the syntax tree. These return/take as an argument a pointer to the node. Typically, a rule creates one node of the appropriate type. Rules have a parameter of type pointer to pointer of the node type, which is used to store the result of the `new_` call in. Rules use a local pointer variables to store results from subrules in. In the following example (LLgen syntax) `rulea` creates a node using the result of another rule, `ruleb`.

An exception to this approach are rules that produce a list. The easiest way to implement these is using two rules, one for the list and one for each individual item. The list rule creates a `List` node, and passes this to the item rule, which appends the item to the list instead of returning it through a pointer parameter.

See `list.h` for details on the available list operations.

A.6 File list

Finally, the following table summarizes the purpose of each source file.

lex.l	Flex source file
lex.yy.c	Flex output file
grammar.y	Bison source file
grammar.tab.*	Bison output files
grammar.g	LLnextgen source file
grammar.[ch]	LLnextgen output files
declaration.* expression.* list.* statement.* type.* scope.* token.* error.* io.* salloc.* bool.h lex.h main.*	Compiler sources
libcb.* rts.* gc.*	Run time system