

## Chapter 3

# Extending XCOM

### 3.1 Overview of XCOM

XCOM is compile-and-go; it has a front-end which analyzes the user's program, a back-end which synthesizes an executable, and a runtime that supports execution. The front end has a grammar module, a scanner, a parser and a symbol table builder. The back end has a tree walker, code generators, an assembler and a manager for register spills. The runtime holds C++ code that can be called from generated machine code. Most of the compiler modules are C++ classes. The design emphasizes conventional algorithms and ease of change over more traditional objectives such as speed or code quality.

There are two independent global variants to XCOM. The first is the target platform. Over the years this has included the IBM, DIGITAL, INTEL and MOTOROLA computers. The second is the parsing method, either top-down or bottom-up. Some variants are implemented in the XCOM distribution. The variant code itself resides in appropriately named subdirectories. Thus the overall structure of the `xcom` directory is:

<code>/xcom</code>	main directory
<code>/fe</code>	analysis (front end)
<code>/cfg</code>	grammar processing
<code>/scan</code>	lex and scan
<code>/parse</code>	parse and ast building
<code>/recur</code>	top down
<code>/lalr</code>	bottom up
<code>/sym</code>	symbol table
<code>/be</code>	synthesis (back end)
<code>/gen</code>	code generation
<code>/x86</code>	Intel x86
<code>/ppc</code>	Motorola power pc
<code>/a64</code>	Intel 64 bit

<code>/asm</code>	assembler/disassembler
<code>/x86</code>	Intel x86
<code>/ppc</code>	Motorola power pc
<code>/a64</code>	Intel 64 bit
<code>/mem</code>	memory pool
<code>/tex</code>	pretty printing
<code>/rt</code>	runtime routines
<code>/bin</code>	xcom binaries
<code>/x</code>	X programs

## Testing

One can test XCOM one module at a time (unit tests), or as a whole. Each unit test is run by a standalone driver. Each driver has some output that illustrates some of the function supplied by the module. The drivers are a good place to exercise new code and place new tests. There are several testing targets in the Makefile. Some of them check that error-free X programs get the expected answer. Others check that mistakes get the expected diagnostic. There are also many assertions insuring XCOM invariants are maintained.

Each major module (e.g., `asm.cpp`) has a `#define/#undef` pair that controls tracing-on-the-fly. Interchanging the lines toggles the tracing behavior. The trace output is to `stdout`; it is useful when one is deep in the details of a module. Typically the (verbose) traces are turned on only during unit testing.

Each major module also has a dumper which presents a readable form of the intermediate information. A dump is not available until the module completes its task. Dumping uses callbacks, passing information to the outer layer of the compiler for presentation. Dumping is requested with XCOM command line flags.

Diagnostics are implemented with exceptions and callbacks, leaving it to the outer layer to decide how to present the comeupance. Runtime errors (e.g. divide by zero) are reported from the assembly code via conventional return codes. A non-zero return code will cause an XCOM error to be thrown following the return.

## Plug and Play Components

Technology changes all the time. Today's best is surely going to be replaced in the near future. The cpu-dependent files are in subdirectories named for the cpu (e.g. `/x86`). Early ancestors of this compiler ran on IBM S/360, DIGITAL VAX and ALPHA. More recently it has run on INTEL X86, POWER PC and INTEL 64-bit cpus. The Makefile detects the underlying hardware and chooses the corresponding subdirectories.

The choice is between top-down and bottom-up parsing is up to the user. A single module, `lang.cpp`, containing the source-language dependencies, has two forms. If the bottom-up (like YACC) form is chosen, LALR(1) tables are built and used. Otherwise recursive methods do the parsing. A `make` variable must be set to select either the `parse/recur` or `parse/lalr` subdirectories.

## 3.2 Changing X Syntax

If one is going to extend x, it is usual to start with the grammar `xcom/fe/cfg/X.cfg`. If something is added to `X.cfg`, then new symbols will be made available for use.

Do not expect too much from the grammar. The grammar already allows some nonsense. For example

```
x + y
```

is not implemented for logical values. Extending the grammar will almost always add some more nonsense in addition to the desired constructs. The undesired constructs must be screened out (cause diagnostics) later in the process, usually when they fail to produce type-correct expressions.

If a bottom-up parser is to be built, the `make` process may halt in directory `/cfg` with diagnostics from the LALR(1) table builder. The table builder must be accommodated via grammar changes to make further progress. If a top-down parser is to be built, the same kind of effort will be expended on getting the recursive routines in `lang.cpp` to work.

### Adding Vector Syntax

Suppose one wished to add vectors to x. The process starts with adding the syntax for `a[i]`. A new rule has to be added:

```
var
  id
  id [ expression ]
```

and the uses of `id` in `factor` and `vars` replaced with `var`.

If one then builds `XCOM`, new tables will be extracted from `X.cfg` and used in analysis and synthesis. The new terminal symbols `[` and `]` will be given names `lsqrSYM` and `rsqrSYM` by `cfg.awk`. You will want to propagate these changes into the lexer, parser, and all the downstream parts of `XCOM`. You can see the tabulated grammar data in `X.macros`.

### Adding Set Syntax

Suppose, on the other hand, one wish to add sets to `XCOM`. This is somewhat more complicated than adding subscripts. One might add additional rules to `factor`:

```
factor
  { exprs }
  { }
  size factor
  choice factor
```

and, assuming that the *relations* (e.g.  $<$ ,  $<=$ ) will be interpreted as the analogous set operations (e.g., include, include-or-equal),  $+$  as set union,  $*$  as set conjunction, one adds just one new *relation*.

*sum* in *sum*

As before, new nonterminals (in this case `lcurlSYM` and `rcurlSYM`) will become available for use throughout the rest of XCOM. And, as before, some syntactically allowed texts (such as set division) will be nonsense.

## Adding String Syntax

Adding a new literal (e.g. C-style string) requires editing the table generator as well as the grammar. One first adds a name

`string`

into the `X.cfg` rule for *factor* analogous to `integer` and `real`. If nothing else is done, `string` will be taken as a reserved word. One needs, in addition, to add `string` to the list in function `isNotReserved` in `lextables.cpp`. This change keeps `string` out of the reserved word list, and enables an extension of the lexer to handle string literals.

One needs some string operators. One can use  $+$  for catenate, as is done in Java, and the relationals for comparisons; these are already in the grammar. Function such as `substr` and `strlen` can be added in analogy to `i2r` and `r2i`.

## Type as Syntax

One can elaborate `X.cfg` to make type-correctness a syntactic issue. For example, there would be separate rules for

em intsum + *intterm*

em realsum + *realterm*

One consequence is adding about 30 rules to `X.cfg`. Another is to force parsing decisions to depend on the knowledge of type. There is such a grammar (`Xtype.cfg`) in the XCOM distribution. It is there as an example of something to avoid, rather than something to emulate.

## Unit Test

The `make` target `cfgtest` will process `X.cfg` and print the information extracted from it.

## 3.3 Changing Lex and Scan

The lexer is based on the assumption that the entire source text of an X file is read and placed in contiguous memory.

The lexer deals directly with memory image of the source text. Anytime a new symbol is added, the lexer potentially needs to analyze a new case. Some situations, such as adding single-character symbols, operator identifiers, and

reserved words, are handled automatically by the grammar processing module. Such input symbols will be given new standard names and passed on to the rest of the compiler without further work by the developer. Input code for new literals, such as strings, needs to be added.

There are three parts to a token: its starting address in the input text, its length, and a code identifying the kind of token. The lexer fills in a C struct and reports its find via a pure virtual function. The scanner implements the virtual function and queues up or discards the result.

The scanner typically does not have to be changed. It is a navigation layer on top of the stream of lexical information.

## Unit Test

The program `scandriver.cpp` contains a set of tests exercising the lexer. It is a good place to add new tests for extensions.

## Dump

The flag `-lex` on the XCOM command line will cause XCOM to dump the sequence of lexemes in a readable format after processing the whole input file.

## Trace

`define/undef` macros at the head of `scan.cpp` can be interchanged and the scanner rebuilt to give a verbose on-the-fly presentation of scanning.

## 3.4 Changing Parse

The parse machinery is defined in `parse.cpp`. It rarely needs changing. The parser itself resides in `lang.cpp`. Its entry, method `parse` implements a pure virtual function defined in `parse.cpp`. The rest of the `Lang` object is private; it reports the shift/reduce sequence to the abstract tree builder via functions of that name defined in the parse machinery.

There are two versions of the X-specific code. One, in subdirectory `/recur` is recursive descent. The other, in subdirectory `/lalr` is bottom-up, driven by LALR(1) tables computed in the grammar module.

In `/recur`, recursive functions must be added for each new phrase name added to the X grammar. The technique for removing left recursion is discussed in the analysis chapter. Function `shift` is called each time a symbol is recognized and used. Function `reduce` is called each time the rhs of a grammar rule is completed.

In `/lalr` the shift/reduce sequence is automatically produced if the grammar was, in fact, LALR(1). If it was not, the build would have broken in the grammar stage.

## Unit Test

The program `parsedriver.cpp` contains a set of tests exercising the parser. It is a good place to add new tests for extensions.

## Trace

`define/undef` macros at the head of `lang.cpp` can be interchanged and the parser rebuilt to give a verbose on-the-fly presentation of the recursive routines at work.

## 3.5 Changing Ast Builder

- tree representation
- node building
- unit test
- ast dump

## 3.6 Changing the Symbol Table

It is reasonable to define vectors to always be “big enough”. That is, if `a[i]` is assigned and there is no *i*-th member, then the array is quietly extended out to position *i*. Trying to use the value of a non-existent `a[i]` would be reported as a runtime error.

- symbol representation
- adding type
- adding shape
- symbol table representation
- unit test
- symbol table dump

## 3.7 Changing Ast Walkers

- postorder walk
- values in walker locals
- opaque locals

## 3.8 Changing Code Generation

- new primitives
- calling runtime
- unit test
- asm dump

dis dump

### 3.9 Pretty Printing

The overloading of other ASCII characters for multiple uses is annoying, both to the author of the extension and to the eventual users of the language. The text  $A + B$  does not immediately bring to mind set union. If the user program were to be typeset, the symbol  $\cup$  could be used, and so on for the rest of the mathematical operators in the set extension. What is even more annoying is the obvious truth that by the end of compilation XCOM has made this distinction. It knows. So provision is made for XCOM to do the typesetting as a byproduct of compilation.

Preceding a file name in the XCOM command line with the flag `-tex` will cause XCOM to dump out LATEX typesetting instructions. The XCOM distribution renders  $x$  according to the choices Dijkstra made in his book *A Discipline of Programming*. The typesetting code can be extended, using the information in the symbol table, and the aesthetics of the language designer, to provide very pretty printing.

### 3.10 Machine Language

Instructions, Registers and Faults

### 3.11 Arithmetic Expressions

Sequencing

Int Register Manager

Left-to-right Float Constraint

### 3.12 Flow of Control

Branches

Fixups

### 3.13 Subprograms

### 3.14 Runtime