Compiler Construction

supplementary notes for Lent Term 2000

A C Norman

Part IB, II(G) and the Diploma

1 Introduction

For the year 2000 the Compiler Construction course will be given partly by Arthur Norman and partly by Martin Richards[1]. The main notes from the course as given by Alan Mycroft are being issued in their normal form: these notes form a supplement and explain

- Which lectures will be given by which lecturer;
- How the order in which material will be presented this years differs from the order in the main lecture notes;
- Details of Java versions of some compiler-construction tools where the main notes discuss C-based ones;
- Minor other areas where there will be changes of content or emphasis this year.

The overall syllabus, and hence the scope of the examination remains just as before: those concerned may consult the laboratory's selection of syllabus statements on its web site. The lectures will concentrate on a particular set of Java compiler construction tools, but the main body of notes (being unaltered from last year) continue to explain lex and yacc (the standard C based toolkit). In such cases where there have been changes it it expected that (a) any examination questions will focus on the version as taught this year and covered in these supplements to the notes, (b) the old version will be discussing something that is intellectually close to this year's coverage and so reading about it is liable to help you not hinder you and (c) regardless of examinations the material in the notes may provide useful backup when you come to apply some of these idea in your later career.
The first twelve lectures will be given by Arthur Norman, who intends to cover:

1. Introduction;
2. Lexical and Syntax analysis;
3. Parser generation tools, specifically `cup` and `jlex`;
4. Simple type-analysis and checking;
5. Language constructs and their implementation.

The remaining eight lectures, by Martin Richards, will cover

6. Survey of execution mechanisms;

[1]Alan Mycroft is on leave.

7. Translation phase;
8. Code generation;
9. Runtime.

For the first part of the course the main books refereed to will be Modern Compiler Implementation in Java[**?**] and the Red Dragon[1] book[2].

A compiler is something that takes the description of a program from a user and converts it into a form that a computer can make much more direct use of. It is useful to think of this translation process as being done in two parts. The first involves un-picking the input files, building neat and rational data-structures to represent the user's intents, checking for consistency and then applying generally valid conversions that simplify or optimise the program. The second part is concerned with mapping high-level constructs in a programming language down onto the concrete operations that a real computer can perform. This may involve making interesting choices about the representation of data, and provides scope for lots of effort to go into the machine-specific aspects of optimisation. By and large the two lecturers this year will split coverage at a point where these two aspects of compilation meet.

Compiler technology is, in various guises, used in many more places than just things that are explicitly labelled "compiler". This means that an understanding of the issues and techniques involved can be useful to almost every computer scientist. The parsing techniques that form the early phases of a regular compiler may be useful when decoding all sorts of structured data: HTML, word processor documents, serialized data being sent as part of a remote method invocation and many more. Lessons from type-checking and other consistency tests applied by compilers can help keep other data structures straight. Almost any textual user-interface will need some sort of parser to disentangle the user's utterances and a study of compilers can let us implement things in an organised and uniform (and hence reliable) way.

Similarly the issues that are central to the codegeneration parts of a compiler have wide uses elsewhere. Page markup languages (eg `latex` and `postscript`) are specialised and rather curious programming languages: knowing about compilers must help in implementing them well. Visual text layout systems will still need to do the same sorts of adjustment and transformations on their data so can use related technology. Some graphics-rendering procedures may be best performed fast by arranging to generate machine code tuned to the particular situation and then executing it: again this is related to the code-generation parts of any compiler.

[2]So known from the picture on its front cover.

Almost any large current system will have an embedded scripting language and handling that is a compilation process. In these varied applications compilation may not be seen as a central part of what is going on, but without knowledge of the established techniques of compiler construction it is probable that seriously slow, clumsy or unreliable solutions will be invented.

2 Lexical and Syntax Analysis

This section of the course is concerned with reading in the file that contains a program and building an internal representation of that program. Almost always it is reasonable to imagine that this representation comes in two parts. The first is a *symbol table* that keeps track of all the names used by the programmer, and the symbols recorded in that take their place in a *parse tree*. This tree will have a node for each construct and each operator in the original program, and its structure will make quite explicit all groupings. Most current practical compilers will build and analyse these structures using three rather separate bodies of code:

Lexical Analysis: This will split the source code up into *tokens*, ie names, numbers, operators and so on. It will discard comments and white-space (ie blanks and newlines). This is usually done as a separate sub-program (a) because it can be, and this leads to a clean logical structure overall, (b) because this task is (amazingly) often one of the most expensive parts of a whole compiler, so using careful code and special tricks in it can be valuable, and (c) because there are convenient tools that can help write the program that does this work;

Parsing: This is sometimes called *syntax analysis*. Very often the compiler designer will select a set of syntax rules that are easy to work with here but that lead to a parser that is somewhat over-generous in that it will accept some strings of tokens that do not form real valid programs: these excess cases are then filtered out later. Proper error detection, reporting and repair are critical issues for parsers;

Checking and Tree Transformations: These must arrange to generate any diagnostics that the earlier steps failed to, and might simplify or otherwise transform the parse tree into some usefully standard state.

There are a number of key ideas that form the foundations of all sensible ways to write these parts of compilers:

1. A formal description of the language that you are trying to accept should be prepared *before* you start writing any programs at all. This means that

a study of compiler construction has to begin with a discussion of the language description notations that have emerged as the common ones used in practise. It also makes it sensible to talk about a rather general categorisation of possible sorts of languages that one may try to use in programming languages;

2. A theoretical basis for processing text against these language descriptions is needed, and the trade-offs involved between exploitation of this and use of ad hoc programs needs to be understood;

3. In almost all cases there will be practical and fairly convenient tools that turn descriptions of a language into a program that will parse it. The benefits and limitations of these and the details of how they are used can be important.

2.1 Lexical Analysis

Amazingly if is often the case that in a real complete compiler a major part of the time is spent reading the input file and splitting it up into tokens. It will generally be necessary to remove comments and categorise input as numbers, symbols, keywords or punctuation characters. It will often be the case that certain pairs of punctuation characters, when occurring together, should be treated as a single item (eg in Java and C the pairs `++` and `<=` (and many more) are treated this way.

One could imagine a programming language with amazingly messy rules about just how characters should be grouped into tokens, and for instance it *might* have been that the Java designers had wanted `a+++++b` (with no blanks at all) to be treated as `a++ + ++b` which is probably the nearest to a meaningful possibility for it. For that to happen there would have had to be some way to ensure that in this case the string of five `+` signs got clustered into a 2-1-2 break-up. Pretty well every modern language designer has taken the view that such treatment is too delicate: one runs the risk of ending up with valid programs that might have two or more interpretations (in the above example one could imagine the first `+` as an infix operator and the remaining four all prefix operators acting on `b`) nd there is a risk of the user being confused as well as the compiler. So more or less universally[3] splitting programs into tokens is expected to be done without concern for context and in a way that can be based on a rather simple program that scans through the input strictly from left to right.

The descriptive tool used for this will be *Regular Expressions*. Those of you who missed out on the Part IA course on them may like to check the notes on my section of the lab web pages (find where these Compiler Construction notes

[3]The main counterexample that springs to my mind is FORTRAN, but a careful discussion of the way in which one might tokenize that really lies twenty years in the past now.

are. . .), obtain a copy of last year's printed notes or read a book. I will not actually need very great depth of understanding. As examples, here are a few (perhaps crude) attempts to use regular expressions to describe the shape of a few of the sorts of token that Java has:

```
decimalint: (1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
octalint:   0(0|1|2|3|4|5|6|7)*
lesseq:     <=
plusplus:   ++
less:       <
keyword_if  if
```

A challenge at this stage is to construct a regular expression that generates exactly the strings that Java would consider to be valid floating point constants. Before doing that you almost certainly have to find a very detailed section in a Java book to find out *exactly* what the rules are.

Those of you who are on the ball will already have notes that I am cheating somewhat: but it is a cheat that is universally accepted! The regular expression I showed above for a decimal integer will match an arbitrarily long sequence of digits. In a real Java program you ought never to see a decimal constant standing for a value more than $2^31 - 1$ for an int or $2^63 - 1$ for a long. However for the purposes of lexical analysis the usual understanding is that these constraints are based on the *meaning* of the string of characters and so overflow will be checked for later on. A typical lexical analyser will treat a sequence of 50 digits as a single token representing an integer and leave it to later parts of the compiler to moan about the number being too big. In practical lexical analysers one may impose some upper bound on the length of a name or number (to allow for the size of buffers used to store things) but it is normal to pretend that this is not so!

Three key things motivate the use of regular expressions as a way of describing tokens:

1. They are easy to write and seem natural and convenient to work with. At least this is true if various extended notations are used: but the well developed theory allows us to be certain that these extensions do not bring any unpleasant side-effects;

2. They appear to be a good natural fit to our intuitive idea of how how text should be split into tokens;

3. A big result, Kleen's theorem, tells us that any regular expression can be converted into the description of a finite automaton that will "accept" items matching the expression. This leads to a good way to implement tokenizers

based on regular expressions: finite automata allow one to recognise tokens working efficiently character at a time.

Using the most primitive notation of Regular Expressions would be possible here but is so clumsy as to be unworkable. The short-cuts most normally used will be

- alternation between various characters can be compressed, for instance `[a-zA-Z]` might be used to stand for any single letter. Expanded out it would have to be written something like `(a|b|c|...|z|A|B|...|Y|Z)`
- The notation `S+` is used to stand for one or more instances of `S` in much that way that `S*` stood for zero or more repetitions, as in `[0-9]+` for a string of digits of length at least one.
- It is sometimes useful to layer on a notation that is much more based directly on finite automata. Such a facility allows the tokenizing engine to be in one of a number of major states. These might, for instance, represent the situation of being in the middle of processing a string, a comment or normal program text.

Anybody who has spent time working through the Regular Languages and Automata lecture notes will be aware that following through the application of Kleen's theorem to derive a finite automaton from a regular expression is somewhat tedious: not something most people would like to do by hand very often. However it is systematic enough that programs to do it are neither large nor especially complicated. I will come back to a description of a use of a particular such tool after I have discussed the next stage of parsing.

2.2 Parsing

Once again the key to parsing involves finding a way to describe grammars. The method that has become totally dominant involves *phrase structure grammars*. The supposition is that the material to be parsed is a sequence of tokens (not just of characters) and that the parsing only depends on the types of these tokens not on any other attributes. For instance if a number can appear at some place then *any* number can appear. Thus an obvious start to defining a grammar is to list all the tokens that can possibly appear. These are referred to as *Terminal Symbols*. On then has another set of symbols, referred to as *Non-Terminal Symbols*. One non-terminal is identified as a *Start Symbol*. The grammar is completed by giving a set of re-write rules. Each of these (in the most general case) has a sequence of symbols (maybe some terminal and some non-terminal) on the left and another such sequence on the right. There must be at least one non-terminal in the pattern in the left hand part of a rule. The set of sentences that make up the language defined by such a grammar can be obtained by starting with the start symbol. Then any time a sequence on the left of a rule appears in the string it can be replaced by the corresponding sequence on the right. If at some stage a string with no non-terminals arises then that string is part of the language. If there is no way of generating a string of terminals, however many productions (the term usually used for what I just started to call re-write rules) are used then that string is not part of the language.

It turns out to be interesting and sensible to consider various restrictions on the form of the productions.

The first such set of conditions looks backwards in this course: insist that the left hand side of each production is just a single non-terminal, and each right hand side is either empty or consists of just a terminal followed by a non-terminal. Eg

$$\begin{array}{lll} \mathtt{S} & \Rightarrow & \mathtt{A} \\ \mathtt{A} & \Rightarrow & \mathtt{a\ B} \\ \mathtt{B} & \Rightarrow & \mathtt{b\ A} \\ \mathtt{B} & \Rightarrow & \end{array}$$

In this example I have used upper case letters for non-terminals and lower case for terminals. Some typographic distinction is generally useful, but just what is used will depend on context. If `S` is the starting state then this grammar generated the language `a(ba)*`. In general grammars restricted in this severe way correspond very naturally to descriptions of non-deterministic finite automate (the non-terminals are states, and where there is a production with an empty right hand side that marks an accepting state).

The next useful class of phrase structured grammars impose just the constraint that the left hand side of each production should be a single non-terminal, as in the example:

$$\begin{array}{lll} \mathtt{S} & \Rightarrow & \mathtt{A} \\ \mathtt{A} & \Rightarrow & \mathtt{a\ A\ a} \\ \mathtt{A} & \Rightarrow & \mathtt{b\ A\ b} \\ \mathtt{A} & \Rightarrow & \end{array}$$

Informal study of this should convince you that the language it defines is that of palindromes over the alphabet `{a,b}`. We know that this language is not regular so no regular expression could define it and no finite automaton could accept it. Grammars where the left hand sides of productions are single non-terminals are known as *context free* grammars. Results you know already say that regular languages can be recognised using finite automata. A very similar style of result asserts that context free languages can be accepted by *stack automata*. These augment a finite-state control mechanism with a stack! Just as there is a pumping lemma for regular languages (and it can be used to show that certain languages are not regular) there is a pumping lemma for context free languages and again it can be used to show that some languages are not context free. The usual example to quote is the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ This consists of the strings `abc`, `aabbcc`, `aaabbbccc` and so on where in each case the number of occurrences of each letter match.

Many texts on parsing discuss a class of grammars known as *context sensitive*. I will not emphasis these here, but will mention the way in which they correspond to a model of computation again later on.

Unconstrained phrase-structure grammars still generate languages. Once again they correspond to a model of computation: in this case Turing Machines. The fact that the ability to parse a general grammar of this form can call on the full power of general computation (and conversely any Turing machine program can have its behaviour captured in a language defined by some phrase structure grammar) tells us that there will be many undecideable problems around and that general grammars will not usually be sensible things to work with.

A final remark is that since there are computations that even a Turing Machine can not perform there will be languages that can not be defined at all phrase structure grammars. The most obvious example will be the language of "all programs in existing computer language X subject to the extra constraint that they must terminate". And of course this would be a grammar we really did want our compiler to accept, so it could generate a diagnostic "this program may not terminate" for input that had that property!

Many years of experience has shown that context free grammars are powerful enough to express the structure that people want in programming languages, but constrained enough to be practical to work with. But just as when I explained that regular languages were used for lexical analysis but that left over issue such as name-length, over-large numeric values and so on, when we use context free grammars for parsing we give up on checking such things as having variables referenced only when in scope, having types used consistently, guaranteeing that the programs we parse will terminate and all sorts of other conditions. Some of these can be checked in later parts of a compiler.

Before looking at detailed parsing parsing techniques there is a slight area of conflict between the language theory approach and the practical one we need. The idea of mapping a grammar onto an automaton works in the context of "accepting the language's. This means it is directly valuable if we want a syntax checker that reports a simple yes/no answer as to whether our input text satisfies the rules of our grammar. However in most real compilers one wants more: a trace of which production rules were fired to get to the input sentence. These can be interpreted as a *parse tree* for the program. Extending a parser to build one of these is typically not that hard, but the big issue that this does raise is that of *ambiguous grammars*. These are one where some particular sentence might be generated on two or more ways. In each case the parsing mechanism will accept the text, but there is no certainty which parse tree will be created. The simple approach to this is to restrict attention to grammars that are not ambiguous. A more complicated scheme allows grammars to start off ambiguous but provides ways of annotating the grammar to ensure that just one version of the parse can actually happen.

A further messy area is that of error recovery. The beautiful theory of stack automata allows us to accept valid programs really rather efficiently. It does not automatically help too much with producing error messages that are much better than

```
+++ Error: your file does not contain a valid program.
```

and this is a bit sad. Again practical parsers will need to extend the theory with practical techniques (and sometimes tricks) to do better.

The main Compiler Construction notes and the Red Dragon book describe various ways for moving from the specification of an unambiguous context free grammar to a parser based on it. In this supplement to the notes I am not going to write all that out again, but I should cross-reference some of the key terms: LL-parsing, Recursive Descent, left-recursion, LR-parsing, shift, reduce, precedence: general precedence and operator precedence, LR(0), LR(k), start-sets, characteristic finite state machine, ALR, LALR. A general message from all that is that there is a great deal of quite complicated technology that goes into understanding and building good parsers! The course expects you to understand what the options are and to be able to apply the techniques by hand to small grammars.

2.3 CUP and JLex

Building any sort of LR parser by hand is tedious. For the grammars of full programming languages it is not really feasible. However the steps involved are all systematic, and so people have written parser generators to do the bulk of the work for you. A notable feature of these is that (for syntax checking at least) there can be a *single* program that is an interpreter for a general stack automaton. The transitions it must make in response to input symbols can all be stored in a neat tabular form. Processing an input symbol then involves taking the current state and the new symbol and looking in the table: what will be found is an action which will be either *shift* or *reduce*. The first of these pushes an item onto the stack, the second pops one or more items before putting a replacement on. In each case the machine will generally change its state. The main parsing loop is thus actually **very** simple and it only performs a rather few instructions as it deals with each new input symbol. Thus parsing is typically rather fast. All the cleverness goes into the preparation of the tables!

If you study a book on parsing techniques you will find that there are several available, notably LR(k), SLR and LALR. These differ in the pain it is to generate the tables, how compact the tables are and how wide a range of grammars can be coped with. Typical real parser generators use LALR which is not the easiest one to understand but counts as a "best buy" in terms of ability to cope with the widest range of grammars for sensible table size.

The software tool discussed here is called CUP, and a version has been placed in the laboratory web pages close to where these compiler construction notes live. When I was preparing this course I came across several different versions of CUP and for now I suggest you use the version I have provided a copy of so we are all certain we are talking about exactly the same thing!

A generated parser will need to call on a lexical analyser to give it a sequence of tokens. The interface between the two programs is a potential area for confusion and although one could in theory interface almost any lexical analyser to almost any generated parser it is much easier if the two start of consistent. Thus as part of the CUP kit that I provide there is a compatible lexical analyser generator called Jlex. In each case the main documentation (including copyright statements) is there as an HTML file. In case of doubt please read the documentation, which I wish to consider as an extension to this handout (again for those of you minded to litigation I should indicate that there will not be examination questions on the minutiae of CUP and Jlex based on material only available in their manuals, but you will find that writing your own parsers using them will cause you to understand them much better (which will be good for both exams and for project work next year and later in life) and the manuals will be helpful then).

To introduce CUP and JLex I will provide the example that comes with them, which builds a very simple desk calculator. This example is just big enough to illustrate the main features. First consider lexical analysis. Here I am going to support a very simple syntax so that the only symbols used are numbers, addition, multiplication, parentheses and a semicolon. I think up textual names for each, for instance `INTEGER`, `SEMI` and so on. Then prepare the following file, which should be called `minimal`lex.:

```
import java_cup.runtime.Symbol;
%%
%cup
%%
";" { return new Symbol(sym.SEMI); }
"+" { return new Symbol(sym.PLUS); }
```

```
"*" { return new Symbol(sym.TIMES); }
"(" { return new Symbol(sym.LPAREN); }
")" { return new Symbol(sym.RPAREN); }
[0-9]+ { return new Symbol(sym.NUMBER,
              new Integer(yytext())); }
[ \t\r\n\f] { /* ignore white space. */ }
. { System.err.println(
       "Illegal character: "+yytext()); }
```

This may look a little dense, and for a grammar as simple as this you might think you could do as well writing the whole lexical analyser by hand. However for bigger examples the use of Jlex will really pay off! There are some section separators involving percent signs, and a directive telling Jlex it will be working with CUP. But then the main body of the file consists of (extended) regular expressions followed by actions to perform when the expression concerned is found. Here most of the regular expressions are simple constant strings, such as `"+"`. In such cases the action is to return something indicating what had been found. The classed `Symbol` and `sym` represent part of the details of the interface to CUP. The pattern `[0-9]+` matches integers, and Jlex arranges that the function `yytext()` can be called to retrieve a string that is the actual set of digits read (and here they are decoded to make a Java `Integer`). The empty action associated with spaces, tabs and newlines causes them to be ignored, and the final pattern (just a dot) is a catch-all that gets activated if no other patterns match.

If you have the set of files I provide you can build a lexical analyser out of this file by saying

```
    java -jar Jlex.jar minimal.lex
```

and you will see a pile of informative messages that may or may not impress you, but you should then find (barring errors) that a file called `Yylex.java` has been created for you. Do not rush to compile it! Until the CUP bit has been done you will not have the `sym` class available and so you will see errors.

To cope with CUP I will work in two stages. In the first I will show a CUP parser that just check syntax and moans if it finds something wrong. Call this one `minimal.cup` and go

```
    java -jar cup.jar < minimal.cup
```

which should make files `parser.java` and `sym.java` for you:

```
import java_cup.runtime.*;

parser code
```

```
{:
public static void main(String args[])
                     throws Exception
{
    new parser(new Yylex(System.in)).parse();
}
:}

terminal SEMI, PLUS, TIMES, LPAREN, RPAREN;
terminal Integer NUMBER;

non terminal expr_list, expr_part;
non terminal Integer expr;

precedence left PLUS;
precedence left TIMES;

expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr SEMI;
expr   ::= NUMBER
        |   expr PLUS expr
        |   expr TIMES expr
        |   LPAREN expr RPAREN
        ;
```

The section labelled `parser code` allows you to include an arbitrary bit of Java in the generated file, and is used here to set up and run the parser. The constructed parser will read stuff and process it when you call its `parse` method.

The next bit lists your terminal and non-terminal symbols. A convention of spelling the terminals in upper case and the non-terminals in lower has been followed, but by declaring all the symbols you make things very nice and explicit. The `NUMBER` terminal is declared to be able to carry back an `Integer` value to you. The others will not carry any information beyond their own type.

Finally there is the context free grammar. The symbol `::=` marks a production, and vertical bars separate alternatives. The grammar as given is ambiguous, but the `precedence` declarations instruct CUP to resolve ambiguities to make multiplication bind more tightly then addition and to make both operations associate to the left. The exact options for and understandings about ambiguity resolution are a mildly murky area!

When you compile all the generated Java code you now have and try it you will get a parser that should do nothing provided you feed it input in accordance with the syntax it supports, but which should moan if you deviate from that.

The next version of `minimal.cup` does a bit more. Each production in the grammar is decorated with a *semantic action*, which is Java code to be executed when the relevant reduction is performed. Items in the grammar can now have names attached (following a colon) to make it possible to refer back to them in the semantic action. Non-terminals can now be given types so that you can do interesting things with them. In this example the non-terminal `expr` is given the type `Integer` (it could not be given the type `int` since valid types here have to be sub-classes of `Object`). Then the actual numeric value can be extracted using the `intValue()` method. You will see that the actions here compute the value of the expression. They could equally well have build a parse tree. In all cases they form segments of Java code enclosed in funny brackets `{:` to `:}` and including an assignment to the special pseudo-variable `RESULT`. Within such a block the names attached to components of the pattern (eg `expr:e` attaches the name `e` to an `expr` that is read) can be treated as items with the type declared for the relevant symbol.

```
import java_cup.runtime.*;

parser code
{:
public static void main(String args[])
                     throws Exception
{
   new parser(new Yylex(System.in)).parse();
}
:}

terminal SEMI, PLUS, TIMES, LPAREN, RPAREN;
terminal Integer NUMBER;

non terminal expr_list, expr_part;
non terminal Integer expr;

precedence left PLUS;
precedence left TIMES;

expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr:e
              {: System.out.println(" = "+e+";");
```

```
                :} SEMI;
expr      ::= NUMBER:n
                {: RESULT=n; :}
          |   expr:l PLUS expr:r
                {: RESULT=new Integer(
                     l.intValue() + r.intValue());
                :}
          |   expr:l TIMES expr:r
                {: RESULT=new Integer(
                     l.intValue() * r.intValue());
                :}
          |   LPAREN expr:e RPAREN
                {: RESULT=e; :}
          ;
```

To try to show that this technology scales I now include a small section of Jlex and CUP input files that I use to define a parser for a subset of the language ML. First an extract from the code that provides a lexical analyser:

```
";"    { return new Symbol(sym.SEMICOLON); }
"op"   { return new Symbol(sym.OP); }
"("    { return new Symbol(sym.LPAR); }
")"    { return new Symbol(sym.RPAR); }
"["    { return new Symbol(sym.LBRACKET); }
"]"    { return new Symbol(sym.RBRACKET); }
","    { return new Symbol(sym.COMMA); }
":"    { return new Symbol(sym.COLON); }
"~"    { return new Symbol(sym.NEGATE); }
"->"   { return new Symbol(sym.ARROW); }
"=>"   { return new Symbol(sym.BIGARROW); }
"|"    { return new Symbol(sym.VBAR); }
"_"    { return new Symbol(sym.UNDERSCORE); }
"nil"  { return new Symbol(sym.NIL); }
"unit" { return new Symbol(sym.UNIT); }
"not"  { return new Symbol(sym.NOT); }
"true" { return new Symbol(sym.TRUE); }
"false"{ return new Symbol(sym.FALSE); }
"if"   { return new Symbol(sym.IF); }
"then" { return new Symbol(sym.THEN); }
"else" { return new Symbol(sym.ELSE); }
"as"   { return new Symbol(sym.AS); }
"and"  { return new Symbol(sym.AND); }
```

```
"fn"    { return new Symbol(sym.FN); }
"let"   { return new Symbol(sym.LET); }
"in"    { return new Symbol(sym.IN); }
"end"   { return new Symbol(sym.END); }
"fun"   { return new Symbol(sym.FUN); }
"val"   { return new Symbol(sym.VAL); }
"local"{ return new Symbol(sym.LOCAL); }
"rec"   { return new Symbol(sym.REC); }
[A-Za-z][A-Za-z0-9_]*
        { return new Symbol(sym.NAME,
                        new String(yytext())); }
[0-9]+ { return new Symbol(sym.NUMBER,
                        new BigInteger(yytext())); }
```

and now a rather smaller (in proportion) extract from the CUP rules about the syntax.

```
terminal String NAME;
terminal java.math.BigInteger NUMBER;
terminal NIL, OP, UNIT, NOT, NEGATE, TRUE;
terminal FALSE, LPAR, RPAR, LBRACKET, RBRACKET;
terminal COMMA, IF, THEN, ELSE, AS, AND, ARROW;
terminal VAL FN, LET, IN, END, SEMICOLON, COLON;
terminal VBAR, UNDERSCORE, BIGARROW, FUN, LOCAL;
terminal REC;
terminal Graph PLUS, MINUS, TIMES, DIVIDE;
terminal Graph REMAINDER, LESS, GREATER;
terminal Graph LESSEQ, GREATEREQ, EQUAL;
terminal Graph NOTEQUAL, ANDALSO, ORELSE, CONS;

nonterminal            program, prog1;
nonterminal Graph      aexp, aexp2, exp, exp1, exp2;
nonterminal Tuple      aexp1;
nonterminal Rule       match, rule;
nonterminal Graph      op;
nonterminal Type       ty;
nonterminal Pattern    pat, pat1, apat, apat2;
nonterminal PatTuple   apat1;
nonterminal Args       fb3;
nonterminal Definition dec, dec1, vb, vb1;
nonterminal Definition fb, fb1, fb2;
```

```
precedence  right      SEMICOLON;
precedence  left       ORELSE;
precedence  left       ANDALSO;
precedence  right      CONS;
precedence  nonassoc   LESSEQ, GREATEREQ,
                       EQUAL, NOTEQUAL,
                       LESS, GREATER;
precedence  left       PLUS, MINUS;
precedence  left       TIMES, DIVIDE, REMAINDER;
precedence  right      ARROW;
/*
 * "->" is specifically more binding than "*"
 * so that a type like   'a*'b->'c
 * will be parsed as     'a*('b->'c)
 */
precedence  left       COLON;


start with program;

program
    ::= prog1
    |   program SEMICOLON prog1
    |   program SEMICOLON
    ;

prog1
    ::= dec1:a      {: Graph.text("Statement ");
                       a.print();
                    :}
    |   exp:a       {: Graph.text("Expression ");
                       a.print();
                    :}
    ;


exp
    ::= exp2:a            {: RESULT = a; :}
    |   exp:a COLON ty:b  {: RESULT =
                               new Typed(a, b); :}
    |   FN match:m        {: RESULT = m.abs(); :}
    |   IF exp:a THEN exp:b ELSE exp:c
                          {: RESULT =
                               Application.ap3(
                                  new If(), a, b, c);
                          :}
    ;


op  ::= PLUS:x            {: RESULT = x; :}
    |   MINUS:x           {: RESULT = x; :}
        etc
    ;
```

I obviously do not want you to understand all the details of the above, especially since it is incomplete. but I hope you can see that in a real(-ish) parser many more non-terminals will be declared to have types (and these will be types as used to represent the parse tree).

2.4 Comparison with lex and yacc

The commonly-used C parser generator tools are lex and yacc (yet another compiler compiler), or their GNU equivalents flex and bison (almost as shaggy as a yak but more American?). Especially if you are tempted to try bison look carefully at the license agreement: the program you end up with will contain its version of the generic parser routines and you may find that if you distribute anything you are obliged to give away all the source code for the rest of your project for free!

The C versions use simple curly brackets to mark out semantic actions. They do not attach types to syntactic categories. Instead of linking names to parts of a patter you have to refer to the parts by number, as in

```
expr ::= expr '+' expr  { $$ = $1 + $3; }
```

where `$$` is used in place of `RESULT` and the `$1` and `$3` relate to the first and third items present in the pattern. In general I think that CUP is somewhat nicer, but if you have learned to use one you can easily enough adapt to the other.

There are a number of other parser generation kits around, and while lex and yacc and quite stably dominant in the C world it may still be too early to know which one will be the ultimate winner for Java. It remains the case that having used one it is very probable that others will not give you trouble.

3 Other parts of the course

Compared with the main printed notes the course this year is liable to have more emphasis on describing compiler issues as they relate to ML and Java. This does not represent a change in coverage, more a slight shift in presentation style. Actually ML and Java between them raise almost all of the important challenges that one can come across in compilers, at least if you apply a little imagination (such as a desire to write Java programs that do massive amounts of floating point calculations very fast).

The main concept that aspects of compiler construction needs to refer to that does not show up well in those languages is that of a *pointer*. This is covered in part in the Computer Architecture courses where they talk about address registers in computer hardware, and it also gets some coverage in *Comparative Programming Languages*. To allow this latter course to have (almost) completed before the compiler construction one needs to make much reference to pointers topics that relate to them will tend to be treated in the second half of the course.

4 Exercises and the like

Both the main text-books suggested here have exercises at the end of each chapter. The computer laboratory has been teaching courses on compilation for over thirty years and so there is a very adequate collection of past examination questions available: transliterating some of the old ones so that they mention Java rather than (say) Algol is itself a valuable exercise. There are also a few exercises in the main notes, and I view it as implicit in almost any computer-related course that keen students should try to apply the technology that they are being taught about in practical cases. So here I can explicitly suggest that you might get the desk calculator example running, expand it to support a fuller complement of operations, then give it memories that it can store values: you may by then be well along the line towards building a simple interpreter for a tidy programming language. As an alternative work towards a syntax checker for ML or Java or C (or whatever) based on CUP but now without having to install any semantic actions.

References

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