1 Introduction and overview of compiler structure

The course is a practically oriented one and tries to mix the application of well established theory with the pragmatics of actual software construction. Little prior knowledge is assumed, but it is essential that you read more than just the notes.

Much of the material follows Aho, Sethi and Ullman's "Dragon book", i.e. *Compilers - Principles, Techniques and Tools*, published by Addison-Wesley. This is very strongly recommended. Other books worth dipping into include

Understanding and Writing Compilers , Richard Bornat, published by Macmillan, a ver yreadable, if somewhat outdated book based on bottom up approaches;

Recursive Descent Compiling , Davie and Morrison, published by Ellis Horwood, written by fans of hand written recursive descent compilers;

The Art and Theory of Compiler Writing , Tremblay and Sorenson, a slightly drier equivalent of the Dragon Book;

Introduction to Compiling Techniques , Bennett, McGraw-Hill, sub-titled "A First Course Using ANSI C, LEX and YACC".

The library has a range of books on compilers and you might wish to browse.

Assessment

The coursework has only two elements, worth 10% and 15% respectively in your overall assessment for the course.

Exercise 1

Starts now and is due in by the end of week seven.

You are asked to read the paper

Pollock and Soffa, April 1992, "Incremental Global Reoptimization of Programs", ACM Transactions on Programming Languages and Systems Vol 14 No 2

and to provide a summary of its contents in 1000-1500 words, including a discussion of the potential usefulness of this approach compared to modular programming.

This need not be formatted with LaTeX. Clear handwritten or ASCII plain text work is quite acceptable.

Exercise 2

Write a bottom up parser, in C or SML, for the language described by the grammar below. You will be expected to produce a running parser on the CS Suns. This should generate output which demonstrates the successful parsing of strings matching the grammar. You should hand in

- a listing of your program or programs;
- output from a set of sample runs, demonstrating the functionality of your parser;

- matching input files, with a note on what each is testing;
- a brief description of your code, which should also be well commented;
- instructions on how your tutor can run the parser;

The grammar

$$\begin{aligned} Expr &\rightarrow Expr\ Arith\ Expr \mid (\ Expr\ ) \mid Id \\ Arith &\rightarrow + \mid - \mid * \mid / \mid \uparrow \\ Id &\rightarrow \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \end{aligned}$$

C language rules of associativity and precedence apply, except for the exponentiation operator, $\uparrow$, which uses Pascal's rules.

Some sample input

```
a+b
a+b*c
(a+b)*c
```

For a reasonable mark, you should recognise, but not necessarily recover from, errors in the input. For full marks, you should report errors clearly and recover to continue recognising subsequent input if possible.

Use of compiler generator tools, such as LEX and YACC, is not allowed.

The work should be completed and handed in by the end of week 9. It is more sensible to hand in a well described, but incomplete, exercise than to spend too long on this work. Work handed in late will be penalised.

Course outline

As with most CS3 courses this one runs for nine weeks. As it has a Monday morning lecture, there is only one lecture in week 1. Unless there is an urgent reason there will be no lecture in week 10, leaving 17 lectures in the course.

All course notes and other material is available through the CS3 World Wide Web pages.

The lectures are planned to be:

1. Introduction and overview of compiler structures. (This lecture.)
2. Analysis stages in a compiler. Introduction of the *symbol table*
3. Systematic lexical analysis - revision of regular expression recognition
4. Syntax analysis and views of grammars
5. Parsing techniques 1 - Top down
6. Parsing 2 - Bottom up, operator precedence parsing
7. Parsing 3 - Shift reduce, SLR parsing
8. Parsing 4 - General LR parsing

9. Synthesis and Semantics
10. High level optimisation 1
11. High level optimisation 2
12. High level optimisation 3 - Loop optimisations
13. High level optimisation 4 - Global data flow analysis
14. The executing program 1 - linking, loading and libraries
15. The executing program 2 - dynamic memory use - procedure calling, heaps and stacks
16. Code generation 1 - Register allocation
17. Code generation 2 - Low level optimisation

Compilers and their structure

A compiler is a piece of software which recognises the structure in sequences of symbols (*strings*) according to a set of rules (a *grammar* or *syntax*) which defines the way in which symbols may be combined to produce legal strings. It then uses a second set of rules (*semantics*) to attach meaning (in the form of *actions*) to the structures recognised. We tend to think of the actions as being code generation from a programming language into machine code, but compilers are widely used with other sorts of language, such as text formatting (LaTeX) and hardware description (VHDL).

Compilers are often divided into phases in ways corresponding to the preferences of the writer concerned or the aspects of its operation considered most significant at the time of writing. Thus a "classical" division would be into phases based on the evolution of techniques for understanding language:

Lexical analysis $\rightarrow$ Syntactic analysis $\rightarrow$ Semanctic analysis $\rightarrow$ Code generation

The early phases here correspond to the development of systematic methods for compiling *context free languages* during the late 1950's and 1960's. Early programming languages were defined without these insights and contained features which made them difficult to analyse lexically and syntactically (phases known jointly as *parsing*) except by *ad hoc* methods. Algol 60, whose definition began in 1958, was the first major language to be explicitly designed to be context free and to have a regular expression based lexical subset grammar. To this day Fortran is struggling to rid itself of its less well defined grammatical inheritance.

As we shall see in later lectures, this classical division is not the only way to think of a compiler. We have already noted that code generation is not the only final activity that might be desired. As the techniques for parsing evolved and merged with the initial stages of semantic analysis and even action initiation, so these came to be considered as the *front end* of a compiler. Much of this change of view came about because the front end could be viewed as *target independent* and so *portable*, while the *back end* and some *intermediate* phases were specific to a machine architecture or printer.

Intermediate forms

As a final point before we begin to examine how compilers actually carry out their tasks, it is important to consider the products at each stage of compilation. Each phase is separated because it is more efficient to perform some aspects of the processing of an input string before others. Thus, lexical analysis is performed before full parsing because it can exploit the regular expression grammar of the language's basic symbols to eliminate inefficient string handling as early as possible and using a simple and fast technique. Its products are a *tokenised* version of the original input, which is much more compact, and a *symbol table* which allows the new tokens to be mapped back to their original strings of symbols.

At each stage of compilation the original input is moved further from its original sequential, string based structure, passing typically through a *parse tree* form before emerging in the target form. *Intermediate codes* may be used at various levels. High level codes are representations of the parse tree. For portability many compilers use low level forms, such as virtual machine codes, which are abstract versions of the final form, such as **P-code**, used in the UCSD Pascal compiler . These allow the target independent development of phases which would otherwise need to be target dependent. Such virtual targets may result in a certain amount of duplication of effort or loss of efficient interpretation when real target output is produced and so some low level codes are even more abstract, remaining closer to the parse tree and symbol table forms used earlier, but making certain high level choices. **DIANA** and **I-code** fall into this category. Still others are themselves directly executable by interpreters, such as the SML *Functional Abstract Machine (FAM).*

The break from the original source in compilation used to occur at code generation, either real code or virtual machine based low level intermediate code. Essentially once the symbol table is no longer passed on, original strings and derived information such as line numbers and textual scope names are unavailable. Yet, with the widespread use of *source level debuggers* and efficient *runtime postmortem diagnostics*, this information is often present even in object files and other target forms.

This leaves the intriguing question of how far the original input string can still be regenerated from its reporesentation at any stage. It also suggests that languages other than the original might be regenerated, as indeed is the case with language to language translators.