Compiler Construction Supervision 2

Matthew Johnson
Trinity Hall
mjj29@cam.ac.uk

March 6, 2003

1 y2000p5q6

1.1 Lex & Yacc

Lex & Yacc are standartd tools for creating lexical analysers and parsers. Lex takes a list of all the possible lexical tokens in a language, and automatically generates code that will recognise these, and convert input characters to the approriate tokens by matching regular expressions. The code normally is created in such a way that is can be called from within the programs output by yacc, to generate the next token each time it is needed by the parser.

Yacc (Yet Another Compiler Compiler) is a tool for creating parsers that will take the lexical tokens produced by the program that Lex generates and output a parse tree that your code generator can create executable code from. Yacc is given a syntatic description of a language in BNF form, and generates code that will match that description. The delcarations for Yacc contain precedence and associativity rules, to allow use of an ambiguous grammar (since thats often easier)

both have the following form in their input files:

```
declarations
%%
translation rules
%%
auxilliary C code
```

1.2 Lex & Yacc input for a parser for given grammar

++

2 y2000p6q6

2.1 code generation

3 y2001p4q4

3.1

S is the start NT, that represents the entire program. The terminals are *';'*, *'{','}'*, *'¡eof¿'*, *'='*, *'¡id¿'*, *'if'*, *'then'*, *'else'* and *'repeatwhile'*. The non-terminals are S, B, C and E.

3.2 ambiguities

C → **if** E **then** C {**else** C}
C → C **repeatwhile** E
to resolve have 3 versions of C, with precedence.
C1 → ¡stuff with C2¿ | C2 C2 → ¡stuff with C3¿ | C3 ...... CN → ¡what C's can go to¿

3.3 abstract syntax

This is asking for a datatype suitable for holding the parse tree - so doesn't matter about ambiguities (we do that in the parser that constructs the parse tree). Use the initial grammar.

```
datatype C = Ass of E*E
           | Repeat of C * E
           | IfThen of E*C
           | IfThenElse of E*C*C
           | Code of C list;
```

3.4 huge SLR/rdP

!! (RdP isn't that big)

4 y2001p6q6

4.1 describe ELF - with eg

The ELF format breaks the object down into seceral distict sections. The first section is the `.text` section, which contains code. Second is the `.data` section - which contains as static data in the program (such as from `x = 37`). Next are 2 sections which contain relocation information for the code and data sections respectively. These are references to the `.sym` table (see later) and where in the code & data sections they should be relocated to, when the external references are resolved.

The `.sym` table contains all the external references made in the object, and what symbols it can export to other objects, followed by a section with these references in string form.

4.1.1 notes

code segment only contains the function definitions, the variable declarations are described by a combination of the data segment, and the string & symbol tables.

the symbol table references variables by their offset in the string table, so all internal and external name are in the string table. The symbol table gives the offset in the string table, the offset into the data or code segments if they are internal, or undefined if it is an external reference.

The relocation table is a link between the code segment & the symbol table, so it a list of pairs of offsets in either table. This is used at runtime to resolve references to data to actual addresses.

4.2 linking

The linker concatenates all the code & data segments of the objects, and uses the relocation tables to replace the dummy values in the objects with the actual values in the new object.

4.2.1 notes

linking:

- code segment: concatenate
- text segment: concatenate
- string table: union
- symbol table: three possibilities. If its in 1 object file, we use that definition, possibly with an updated offset. If its definied in one object file, and undefined in the other, then we take the defined one. If its undefined in both or defined in both then throw an error.
- relocation table: all the offsets will have changed, so update them all.

4.3 static vs dynamic

static: everything you need for the program in the executable. Updating libraries won't break program. dynamic: allows sharing of code between programs, without duplication, and also allows for easier fixing of bugs, when lots of peple use a library - but have to be using the right version.

4.3.1 notes

At link time the object is linked with a *stub* of the library, and then at run time the actual library is loaded.