5 Compiling

5.1 Introduction

Writing a compiler has a reputation of being among the more complicated of programming tasks. This view is dated, however, as in the last decade many software tools have been developed that take care of the more difficult and tedious sections. In the case of the shader compiler writer, the job can be further simplified because the target system which runs the shaders is typically a virtual machine. This chapter will concentrate on the special problems in compiler writing that are particular to writing a compiler for a renderer, while covering enough general compilation topics to get the novice compiler writer off to a very good start.

As with any computer program, there are an infinite number of ways to put a compiler together. The architecture described here has been found to be flexible, extensible and simple to implement.

Simply put, a compiler takes source language and outputs machine code. In many cases, the compiler writer will have no control over the source language – in the case of a shader compiler the choice may have already been made to use RenderMan Shading Language, HLSL or some other standard. Likewise, the target machine may already have been selected. In other cases, the compiler writer may have some input in defining the source, the target or both. In renderers it is common to use a virtual machine to evaluate the shaders (although other possibilities include compiling them to native libraries or for specialized hardware), in which case it may be possible to design a target system to simplify the compilation.

Figure 5.1 shows the major stages of any sort of compilation. Parsing is the input of the source code, and its conversion into the compiler's internal format. Code generation is the output of this internal data as target code. The intervening stages are almost optional – simplification is the "massaging" of the code to make later passes easier to implement, and optimization is the completely optional step of changing the code itself to make it more efficient at run time.

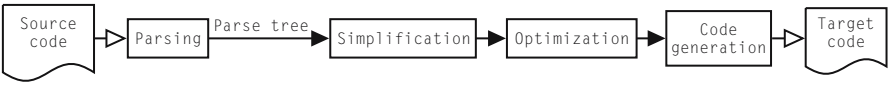

Figure 5.1 The major stages of compilation.

5.2 Data Structures

Most of the intermediate data will be kept in some sort of parse tree. We can define a data structure to support this as:

```
typedef struct {
  Symbol type;
  int length;
  union {
    Node *children;
    Value leafvalue;
  } u;
} * Node;
Node NodeNil();
Node NodeLeaf(Value val);
Node NodeC1(Symbol type);
Node NodeC2(Symbol type, Node);
Node NodeC3(Symbol type, Node, Node);
Node NodeC4(Symbol type, Node, Node, Node);
Node NodeC5(Symbol type, Node, Node, Node, Node);

Node NodeGetNth(Node, int i);
void NodeSetNth(Node, Node, int i);
Symbol NodeGetType(Node);
Value NodeLeafValue(Node);
int NodeIsLeaf(Node);
int NodeIsNil(Node);
```

5.3 Overview

A common mistake when writing a compiler is to try to do everything at once – reading in the source and outputting the code almost at the same time. A far more sensible, and easier, way to do it is to tackle one thing at a time, making many passes over the intermediate data. With this principle firmly in mind, anyone can build a compiler from scratch.

Consider the following pseudo code that illustrates the `main()` method of our compiler:

```
compilerMain (char* infile, char* outfile) {
  Node tree;

  Parse(infile, &tree); /* creates a parse tree from infile
                        and puts it in tree */
  Simplify(&tree);      /* traverses tree, modifying it in
                        place */
  Optimise(&tree);      /* traverses tree, modifying it in
                        place */
```