Design Document

This document will attempt to explain the design of the OddBall compiler and justify our group's design decisions.

1. Output to C

We chose our final output to be C source code. This was done because it eliminated the need to create a runtime environment. It also made our compiler easily portable as the compiler output was completely machine independent.

2. Flow of Execution

The main control of our compiler comes from the parser. The syntactical analyzers asks the lexical analyzer for tokens one at a time. Using a recursive-descent parsing technique for everything but expressions, the parser will validate each line. Along with validation the parser will pass information to the semantic analyzer to make a series of steps. These steps are then optionally passed to the optimizer and then to the code generator. We chose to make all the steps before performing any code to avoid intermediate code before optimization. This is the basic flow of control in the OddBall compiler. The remainder of this document will focus on our key design decisions.

3. Parser Control

A parser controlled compiler simplifies the design of recursive-descent and Samelson-Bauer. Since this is the first compiler most of us has written, we wanted a simple design.

4. Samelson-Bauer for Expressions

To parse expressions, we use the Samelson-Bauer algorithm. This method was taught in class and was simple to implement. It also readily handled unary operators and other special operators.

5. Recursive-Descent Topology

A recursive-descent parser was chosen to parse everything because it was easy to implement. We could easily break the BNF language definition into chunks which could be handed off to individuals. An individual could then translate directly from their section of BNF to code.

6. Steps

This is one of the most unique parts of our design. Steps are basically a form of intermediate code generation except in the memory. Each complete statement was translated into a series of one or more steps. These steps were then collected in a linked list. This helped significantly in the code generation. At the code generation stage, each the step can be easily converted to a simple fragment of C source code.

7. Single symbol table

We chose to use one integrated symbol table with a unique handle for each token. The symbol table handled scoping internally so it was not required to have separate symbol tables for each function. This way, every token could be uniquely referenced by a very simple type. To obtain detailed information about a token, one merely had to provide the token in a query to the symbol table. By having a single interface for every token, the design removed potential complexity from the syntactic and semantic analyzers.