

CSC488
ASSIGNMENT 5
CODE GENERATOR

DANIEL BLOEMENDAL
OREN WATSON

CONTENTS

1. Instructions	1
1.1. Bounds checking	1
1.2. Code dump & syntax highlighting	1
2. Design	1
2.1. Overview	1
2.2. Coalescing minor scopes	1
2.3. Calling convention	3

1. INSTRUCTIONS

Before delving into the design of the code generator, it deserves noting that there are a couple of features that go somewhat beyond the assignment requirements and are therefore disabled by default. We will start with a brief discussion of the first of the two, bounds checking.

1.1. Bounds checking. Disabled by default, the code generator has the ability to add two different levels of array bounds checking. The first level is “simple” bounds checking. It can be enabled with the compiler flag “-B s”. In both modes array bounds are checked on array access. However, in the simple mode when an error occurs a simple error message with limited location information is provided. The second level is “enhanced” bounds checking, enabled with the compiler flag “-B e”. It differs from the second mode in the level of detail provided in the location information contained in the error message. An example of a out of bounds error message is provided below.

Listing 1 Out of bounds error

```
db@vmbox ~/workspace/CSC488/A5 $ ./RUNCOMPILER.sh -B e testing/fail/runtime_bounds2d.488
...
Start Execution  pc = 0, msp = 311, mlp = 16383

testing/fail/runtime_bounds2d.488:7:5: subscript out of range
    a[1, 11] := 1
        ^^
...

```

1.2. Code dump & syntax highlighting.

2. DESIGN

2.1. Overview. The overarching theme in the design of the code generator was to avoid exposing the code generator class `CodeGen` to the complexities and finer details of the underlying machine. To that end, an assembler was developed that hides the complexity of addressing code via a label system and provides an enhanced instruction set, simplifying the emitted code in `CodeGen`. It should be noted that the assembler is also entirely decoupled from the rest of the code generator and stands on its own. The assembler is covered in more detail in `doc/ASSEMBLER.pdf`. We made a distinction between ‘major’ and ‘minor’ scopes. A major scope is a whole program, a function, or a procedure, requiring its own stack frame; all other scopes are considered minor scopes. The complexities of managing major scopes, their displays, and ensuring that minor scopes are merged into their enclosing major scopes, are dealt with by the `Frame` and `Table` classes.

2.2. Coalescing minor scopes. One of the more involved parts of developing the code generator was to ensure that minor scopes were coalesced into the enclosing major scope. This is rather important as it avoids draining the rather limited resource of display registers, 16 in total. In addition, it reduces the size of the generated code as it avoids having to generate a prolog and epilog for every minor scope. The main class responsible for managing this is `Frame`. It lays out the combined locals of all minor scopes in a given major scope. It does so via an algorithm identical to the record layout algorithm discussed in class. A frame can be thought of as a record corresponding to a major scope where all minor scopes are sub-records. One important detail is that sibling minor scopes can be thought of as belonging to a union. The reason this is a reasonable thing to do is

because only one sibling minor scope is alive at any one time. Therefore, it is safe to overwrite the locals used by any previously executed sibling minor scope.

We will now proceed to an example. We will define a small program and lay out the locals according to the algorithm in **Frame**.

Listing 2 Major scope in 488

```
{
  var a : integer a := 0
  { var b, c : integer b := 0 c := 0 }
  { var d, e, f : integer d := 0 e := 0 f := 0 }
  { var g : integer g := 0
    { var h : integer h := 0 }
    { var i, j : integer i := 0 j := 0 }
  }
}
```

We will proceed to lay out the locals. We will express our layout using a C structure.

Listing 3 Layout as C structure

```

struct major_1 {
    short a;
    union {
        struct { short b, c; } minor_1;
        struct { short d, e, f; } minor_2;
        struct {
            short g;
            union {
                struct { short h; } minor_4;
                struct { short i, j; } minor_5;
            } _siblings;
        } minor_3;
    } _siblings;
};

```

2.3. Calling convention. A key issue that we debated at length was the method by which function and procedure calls would be implemented. We decided that the most important thing in functions was for the calling code to have the result value at the top of the stack when it returns, with no cleanup, so that the result could immediately be worked with. Therefore, the function cleans up its own arguments and places its result into a reserved place at the bottom of the stack frame. We also decided that to avoid undue complexity, we would treat procedures identically to functions, even if this resulted in wasting space with an unused result value space.

To call a function, the calling code reserves a place for the result, pushes the return address and arguments, and then jumps to the function code. The function code then saves the display pointer for its level, sets the display, and allocates space for its locals. When returning, the function deallocates its locals, restores the display pointer to its previous state, pops its arguments off the stack, and finally returns, leaving the result as the only thing left. The frame therefore has the following structure, shown in figure 1, with the stack growing down. It should be noted that the base address $D[LL]$ refers to the display set during the prolog of a function via the `SAVECTX LL` intermediate instruction. Here, LL refers to the lexical level of the major scope.

FIGURE 1. Stack frame

Result	$D[LL] - N - 3$
Return address	$D[LL] - N - 2$
Argument 1	$D[LL] - N - 1$
...	...
Argument N	$D[LL] - 2$
Previous display	$D[LL] - 1$
Locals	$D[LL]$