CSE304 : Compiler Design : Semantic Analysis and Intermediate Code Generation

Phase IV : Semantic Analysis and Intermediate Code Generation

Due: 29-May-2025

Overview

The fourth phase of developing a compiler for ***rascl*** is writing a semantic analyzer and generating intermediate code. How you implement this depends on the type of parser you wrote. For a recursive decent parser, it involves mainly adding new code to the current functions. The functions also must accept parameters for 'inherited attributes' and they must return values that are 'synthesized attributes' for the associated non-terminal.

Semantic Analysis

The semantic analysis must include type checking and the generation of conversion operators (type coercion) in order to generate operations that produce the correct results. Type coercion follows rules similar to C. That is, integer operations are used between integer operands, float operations are used for float operands. In the case of a mathematical operation between an integer and a float, the integer must be 'promoted' (that is converted) to a float and a floating point operation must be used.

Also, the language 'short circuits' logical tests. So in the case of compound tests like:

(a > 10) && (b < 0)

If the first part '(a > 10)' is false, we know the entire expression is false and we do not evaluate '(b < 0)'.

Array references involve calculating offsets and adding these to a 'base' location for the array. RIF provides a way to load an address into a register (or temporary). RIF also provides some 'indexed' instructions for loading and storing so you can take advantage of those to generate less instructions.

Intermediate code format

The Intermediate code you generate will be three address code in the form of quadruples. They will have an operator, two source operands and a result field. Operands for arithmetic operations are always registers. There are an infinite number of registers. Temporary register names for integer values will be T# where '#' is an integer. For floating point numbers, temporary names should be of the form FT# where # is an integer. RIF is described completely in The RIF_Developer_Guide provided.

Final Results

The semantic analysis and intermediate code generation code should be integrated with your other components (symbol table manager, lexical analyzer, and parser). The resulting compiler should be able to generate intermediate code correctly for any of the Rascl programs in the test suite I give you.
I will provide notional examples of code generated for the test suite. The examples include comment statements that indicate where different code starts like *if* statements, *while* statements, *assignment* statements, etc. Your code need not generate these comments (although you may want to try to write these into your code as a help to debug your code generation!)

Testing

I will provide a moderate set of test code for you to compile. I may use additional tests when I grade your intermediate code generation. Here are a couple of short examples of what RASCL code looks like. Two are very basic tests using minimal features. The third includes function definitions and calls as well as multi dimensional arrays. Sample RIF intermediate code for these examples is attached to the assignment dropbox. The code is 'notional' in that it should represent an accurate translation of the Rascl program. However, the code your compiler generates does not need to match this as long as the code translates the source accurately according to the syntax and semantics described in the assignment.

=====
T71_Functions.rsc
=====

```
int a;
int b;
int theSum;
int bignum;

function dude(int a, int b)
{
```

```
   theSum = a * a + b * b;
   return theSum;
}

{
  a = 5;
  b = 10;

  bignum = dude(a, b);
  print bignum
}
```

=====

T71_Functions.rso (notional output of compiler)

=====

```
.segment, 0, 0, .data
.int 0, 1, a
.int 0, 1, b
.int 0, 1, theSum
.int 0, 1, bignum

.segment, 0, 0, .text

.label, 0, 0, L1
lw, T0, 0, (B-8)
lw, T1, 0, (B-12)

la, T2, 0, theSum
lw, T3, 0, T0
lw, T4, 0, a
mul, T3, T4, T5
lw, T6, 0, b
lw, T7, 0, b
mul, T6, T7, T8
add, T5, T8, T9
sw, T9, 0, T2
lw, T10, 0, T2
ret, 0, 0, T10

.label, 0, 0, L2
la, T11, 0, a
li, T12, 0, 5
sw, T12, 0, T11
la, T13, 0, b
li, T14, 0, 10
```

```
sw, T14, 0, T13
param, 0, 0, a
param, 0, 0, b
call, 2, T10, L1
sw, T10, 0, bignum
lw, T15, 0, bignum
syscall, 2, T15, 0
```

=====

The minimal set of test cases is attached to the dropbox as the file *testsuite.zip.* There are two subdirectories. The *basic_rascl*.zip file has tests that any compiler should be able to handle.

Test Output

The compiler will accept a file with an extension of .rsc and produce the intermediate code in a file with an extension of .rso. You should no longer but productions out to the console. The output file will have code compliant with the *RIF Developer Guide* which is also attached to the assignment dropbox.

Submission Process:

The deliverables for this part of the project are:

1. Source code to the entire compiler
2. A script that builds and links the code (if it is in a compiled language like C).
3. A Dockerfile that builds an image including your source code and then loads any tools that will be needed to build and run the compiler. The Dockerfile should also include the creation of two directories (/app/tests and /app/output). I will use those to connect to folders on my machine where I can read the test inputs and write the test outputs. The Dockerfile should also copy in a script that builds and links the code (if it is in a compiled language like C).
4. In addition to your code, Dockerfile, etc, please add a README.txt file that describes how to build and run your code.

Now…

1. A Place the deliverable source files for your compiler [including updated lexer, updated parser if fixes were needed, symbol table manager and other needed modules] This should also include your Dockerfile to create the required virtual container) into a folder by themselves (if you wish, you can place the source code in a folder called src below this top-level folder). The folder's name should be CSE304_Phase4_<yourname>_<yourid>. So, if your name is Cameron Ross and

your id is 12345678, the folder should be named ‘CSE304_Phase4_CameronRoss_12345678.

2. Compress the folder and submit the zip file.
    a. On Windows, do this by clicking the right-mouse button while hovering over the folder. Select ‘Send to -> Compressed (zipped) folder’. The compressed folder will have the same name with a .zip extension. You will upload that file to the Brightspace.
    b. On Mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Brightspace.
3. Navigate to the course Brightspace site. Click **Assignments** in the top navbar menu. Look under the category ‘CompilerProject’. Click **Compiler_Phase4.**
    a. Scroll down and under **Submit Assignment**, click the **Add a File** button**.**
    b. Click **My Computer** (first item in list).
    c. Find the zip file and drag it to the ‘Upload’ area of the presented dialog box.
    d. Click the **Add** button in the dialog.
    e. You may write comments in the comment box at the bottom.
    f. Click **Submit**. ← Be sure to do this so I can retrieve the submission!

***** Late Assignments will NOT be accepted, and any assignments emailed to the professor will also NOT be accepted. Start working early and plan to submit working code on time. *****