Midterm CSE 401, Fall 2001

Robert R. Henry

November 2, 2001

1 General Questions

1. Compile time is so-called static analysis, and is when the compiler executes analyzing the static (unchanging) structure of the program. Run time is when the program being compiled actually executes.

2. The analysis part of the compiler consists of phases that analyze the original source program and break it down into an internal representation of the program. These phases include lexical analysis, syntactic analysis semantic analysis, and parts of the improver/optimizer. The synthesis part of the compiler comes after the analysis part, and is responsible for constructing an equivalent program for the target machine. These phases include parts of the improver/optimizer, the code generator, the assembler and the linker.

3. You would like to have 100% confidence that the compiler does not introduce bugs into the application program. You might use formal techniques whenever possible (such as compiler compilers and table driven techniques). You might want to prove that your compiler is correct. You can increase the robustness of your compiler by keeping it simple. You would also want to have the compiler do static analysis of the source program to ensure that variables are initialized before used, and so forth. You might consider using a run-time system that ensures some kind of security and run time type safety, perhaps emulating what the Java runtime does.

4. Chaos. When it comes time to merge their software they will (probably) not have any tools to help them to a controlled merge, where both Randy's and Kurt's changes are seamlessly folded into one.

5. Advantages: You can leverage off of existing mechanisms (the CFG and the parser for that CFG). Disadvangates: The grammar will get (much) larger. There might be ambiguities in the grammar, or the grammar might be unparseable with your parsing method of choice.

6. How deep does the parser's call stack get when it parses the following sentence with the previous expression grammar?

```
a + (b*c) * d
```

Draw the parse tree. Measure the depth by inspection.

```
E --> T + T
  T --> F
    F --> a
  T --> F * F
    F --> (E)
      E --> T
T --> F * F
  F --> b deepest point
  F --> c deepest point
    F --> d
```

So, the deepest the call stack gets is 6.

7. signature2(P1) = first2(B) = "a" concat follow1(B) = "ax" signature2(P2) = first2(D) = "ab"

8. An item is a production in the CFG with a "." (dot) someplace in the right hand side. An item encodes the position the parser is in when recognizing that production. An itemset is a set of items. So, an itemset encodes the position the parser is in when simultaneously recognizing the right hand sides of 1 or more productions simultaneously.

9.
```
S --> L
L --> ( { L } ) ; 0 or more occurances of an S
L --> epsilon ; nothing
L --> Token ; token is anything except ( or )
```

10. The lazy algorithm can wait until it has as much information as possible and then commit to a decision based on that wealth of information. The early algorithm is likely to be simpler, since it doesn't have to keep track of as much information.

11. An inherited attribute is semantic information propagated down the parse tree. A synthesized attribute is semantic information propagated up the parse tree from leaves to the root.

12. It will never be clear that the implementation is consistent with the specification, since the specification is so fuzzy. The implementation will be come the de-facto standard for the language.

13. For: a hand written scanner is likely to be tuned for the job at hand, and so is more likely to be faster than the table approach. A hand written scanner more likely to be extendable to do (limited) semantic analysis, and be extended to handle situations that aren't easily specified using regular expressions. Further, you won't have to worry about bugs or limitations in LEX itself, just bugs in your own coding. Against: you may find that your own scanner is more brittle, more monolithic, and harder to change to handle some kinds of extensions. You can rely on LEX to generate a compact tabular representation of the finite state machine.

14.
- The most natural place to do this is in the scanner. You can consume the characters making up an identifier, buffering up the incomplete identifier as you skip white space and process comments. If you see a stand-alone sub-identifier that is a keyword, then you are done scanning the identifier. You'll return the value of the identifier (eg, its string-valued name) to the parser, and remember that you have a keyword as well, which you'll return to the parser the next time it calls you. The character buffering and keyword will be the most complicated part of this solution. Integers would be treated in much the same way.
- You can also do this in the parser. The scanner would work as it does now (no change!). Change the grammar so that everywhere an identifier is expected you now expect an identifier list of length 1 or more. The semantic action associated with parsing an identifier list would be to concatenate the string-valued sub-identifiers into a single identifier, and then proceed as it does now. Integers would be handled in much the same way. The advantage of this scheme is that you use the parser to do the buffering, and can let the scanner just deal with white space and comments the way it currently does.

15.
```
procedure convert (length_mm: integer);
  var delay_fs:interval;
begin
  length_mm := input;
  if length_mm < 1000 then
    throw 0;
  end;
  delay_fs := (2000 * (interval(-5,5)+100) * (interval(-2,2)+length_mm))/100;
  output := lb(delay_fs);
  output := ub(delay_fs);
end convert;
```

The factor 2000 is the basic propagation delay, no uncertainty. The factor `interval(-5,5)+100` gives me the plus or minus 5% uncertainty on the speed of light, but scaled by a factor 100. The factor `interval(-2,2)+length_mm`

is the length (read from the input) with plus or minus 2mm of length uncertainty. The entire product is divided by 100 to convert the previous scaling back to fs.