COMPUTER SCIENCE 322 (Winter Term 2004)
Compiler Construction
Prof. Levy

Problem Set 6

Due Monday 08 March

Reading Assignment: Dragon Book 4.7; 7.6

Written Assignment

Part I: Dragon Book exercise **4.33 (e), (f)**

Part II: Recall our grammar for `let` expressions in ML, reproduced below with additional rules for arithmetic and boolean expressions:[1]

> *expr* → *let-expr* | *arith-expr* | *bool-expr* | **(** *expr* **)** | **id** | **number** | **boolean**
> *let-expr* → **let** *bindings* **in** *expr* **end**
> *arith-expr* → *expr* + *expr* | *expr* * *expr*
> *bool-expr* → *expr* **and** *expr* | *expr* **or** *expr*
> *bindings* → *binding* | *binding*; *bindings*
> *binding* → **val** *var* = *expr*

with the usual assumptions about precedence ($* > +$, **and** > **or**), and the following regular definitions:

> **id** → [A-Za-z] $[\text{A-Za-z0-9}]^*$
> **number** → $[0\text{-}9]^+$
> **boolean** → **true** | **false**

Suppose someone makes the following suggestion: for this sort of simple programming language, we can avoid writing a distinct type-checking component, by incorporating type information into the grammar itself. For example, to prohibit mixing up booleans and numbers, we could move **number** from the RHS of the *expr* rule to the RHS of the *arith-expr*, and move **boolean** to the RHS of the *bool-expr* rule.

What is wrong with this argument? Specifically, what crucial element does the solution fail to handle?

Turn your answers in to me on paper.

[1] in reality, ML uses **andalso** instead of **and**, and **orelse** instead of **or**, but I like the simpler versions better.

Programming Assignment

This assignment has two purposes: (1) familiarizing yourself with building parsers using the JFlex/CUP alternative to lex/yacc; (2) trying out different symbol table implementations to see which are most efficient. To complete the assignment, perform the following steps:

1. Unzip the file `ps6.zip`
2. Write a CUP grammar called `TinyML.cup`, based on the grammar in the written assignment. Your grammar should include the precedences mentioned above.
3. Complete the lexical rules for the JFlex scanner in `TinyML.flex`, using the appropriate `symbol` methods for the return values (all three can have hollow methods to start):
4. Following the `SymbolTable` interface, implement three different versions of a symbol table:
   (a) `LinearSymbolTable`, which uses a simple list (*e.g.*, `java.util.Vector`) to store the symbol table entries.
   (b) `PJWHashedSymbolTable`, which uses a hashtable based on the PJW hash-function algorithm in Fig. 7.35 on page 4.26 of the Dragon book. *Hint*: use `long` instead of `unsigned` (which Java doesn't have), to avoid numerical overflow.
   (c) `SunHashedSymbolTable`, which uses Sun's `java.util.Hashtable` (one-liner's for each method you implement).
5. Use the `gentest` program provided to generate sample inputs for your parser. Running the command

   % `java gentest` N > *outfile*

   will put a little ML program with N randomly-named variable declarations into the file $outfile$. You can then test your parser on this file by running the command

   % `java TinyML` *outfile*. Compare the time taken by the three symbol table implementations, using enough different values of N.
6. Turn in the following items:
   (a) `TinyML.cup`
   (b) `TinyML.flex`
   (c) `LinearSymbolTable.java`
   (d) `PJWHashedSymbolTable.java`
   (e) `SunHashedSymbolTable.java`
   (f) Either on paper or in a file, a graph or table showing your results from the last step.