UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Prof. R. Fateman

Fall, 2005

CS 164 Assignment 3 and 4: Parsing for MiniJava

Due: Tuesday, Oct. 13, 2005, 11:59PM (assignment 3), Oct. 20, (assignment 4)

This assignment requires you to examine and extend one parser and use a parser-building tool to program another parser. The first parser is a recursive-descent parser written in Common Lisp: a top-down parser without backup. The second parser, also in Common Lisp, will be produced by an LALR syntax-directed parser generator.

As you should realize by now, programming a top-down predictive parser is often a straightforward and intuitive task. While it is considered especially suitable for small languages, this technique can also be used for rather ambitious programming languages. In particular, once you have written some stereotyped recursive programs for common styles of grammar productions, such a parser can be extended fairly simply. And you can do most of the extension by cutting and pasting. For support, it requires nothing much other than a routine to read tokens (presumably done by your lexical analysis module). Almost any modern language is suitable for writing such a parser, requiring only support for recursive function calls. Writing such a parser in Lisp is especially appealing since our emacs/lisp system provides an environment for creating and testing such a parser "incrementally." With such a programming approach it is relatively simple to identify and recover from many errors, or to informally extend the grammar by allowing certain kinds of corrections: adding in missing punctuation and giving very specific warnings or error messages. It does not require learning "extra tools" such as a parser generator. Not to say such tools should not be used, just that under the best of circumstances they tend to produce mysterious programs, and some of the programs force you to learn and debug "specifications" in yet another language.

On the other hand, as software tools go, parser generators are easy to find, (or build!). LL(1) top-down parser generators are not as commonly used as bottom-up LALR, mostly because typical programming language grammars may have just a few constructions whose simplest grammar is not LL(1). Recall that LL parsers may require rearrangement of the productions in the grammar (factoring and left-recursion removal), making the programmer's job somewhat clumsier: it is necessary to attach meaningful "augments" or "actions" to reductions. In fact our sample MiniJava parser (courtesy of David Bindel) is an LL(1) formulation showing that MiniJava is susceptible to LL(1) parsing.

These days, for a programming language design to be considered respectable, the designers must show that there is grammar for the language that is entirely LALR(1). Occasionally there may be a good reason for a non-LALR(1) construction, but it would be a cause for apology. You will probably find that a grammar, once it is made non-ambiguous, is mostly LALR(1). In the few cases where there are non-LALR spots, or even ambiguities, we can often ignore the warnings: the parser can be set up to automatically do the "correct" thing anyway, in a way analogous to the "maximal munch" lexer rule that does the right thing there.

LALR generators are fairly easy to use, they produce fast and (reasonably) compact parsers, and (for the theoretically minded) can parse a substantially larger abstract class of language constructs than the top-down methods. The use of automated parser generators forces you to use *formal* rather than *ad hoc* descriptions of your language and provides a more direct route from grammar design to implementation. That is, in the face of continuing experimental changes to a language, a parser generator may be quite handy since your "turn-around time" to try each new grammar for consistency requires only the time to edit the grammar and "remake" the parser. This may require only a few seconds. If the "remake" succeeds, not only have you confirmed the formal properties of the grammar, but you have a parser. You can also test fragments of a total grammar in pieces and then put them all together.

Of these parser-generators, widely used systems include the ancient YACC "yet another compiler-compiler". The equivalent BISON, available as "GNU" software, does not require licensing, and has been used in CS164 in the past. The text discusses a free Java tool called SableCC which appears to resemble YACC with minor differences. (JavaCC, also mentioned, is an LL(1) generator.) There are several LALR parser-generators written in Common Lisp. The one we used in Fall, 2002 in CS164 was written by Mark Johnson of Brown University in 1988. It is free, and its source is on the class home page software directory. We've used it in several classes and appears to be free of bugs. It provides some diagnostic information on request, and it produces as output a reasonable (Lisp language) parser for the grammar you supply. You could, if you wish, look at the code and see the pattern that makes it rather like a (sparse) table.

For this assignment you are going to consider use *both* techniques: recursive descent and an LALR parser generator.

1. In the class `software/source` directory are several versions of recursive descent parsers for the grammars in the text, 3.11 and 3.15. Read and understand them in sequence, `recdec.cl`, `recdec315.cl`, `recdec315d.cl`, and finally `recdec315s.cl`. Your task is to change `recdec315s.cl` so that it treats a larger grammar. This grammar should allow for assignment statements of the form `id := expression` and sequences like { `expression , expression, ...`}, and the operators > and <. Note that `a+b>c+d` means `(a+b)>(c+d)` not `a+(b>c)+d` or some other variation.

It would be good to make sure you have a clear idea of what is now legal, so you should write some grammar rules before adding these features to the parser. How do you deal with `a:=b:=c` or empty sequences {}? Note that these parsers have a trivial lexical analyzer, namely `eat`.

2. In the `software/build` subdirectory, there are (compiled) copies of the parser generator `lalr.fasl` for various systems. We have also provided a collection of macros, `make-lalr.fasl`, which provide a more compact interface to the `lalr` parser generator. The source code and documentation for the parser generator is in `software/source`. This source file also includes examples for you to get started. There are many additional example grammars in a form suitable for use with this code (with our wrappers) in the file `jy.lisp`. This source file also includes more documentation.

You are free to use your lexical analysis program or ours. While we believe `lex.fasl` (= `longerlex`) works satisfactorily, you may prefer your own work.

To use `lalr`, start up Lisp and load `lalr.fasl` and some `lex.fasl`. The `lalr` functions are declared inside of a package; to use this package, type `(use-package :lalr)` at the command line. You can then read in `make-lalr`, if you wish, and your particular grammar. For example, the following sequence of commands loads everything you need to try one of the infix expression grammars in `jy.lisp`:

```
(load "longerlex")
(load "lalr")
(load "make-lalr")
(use-package :lalr)
(load "jy")
(eval (jparses "1+2*3"))
```

Your warm-up task is to extend the grammar (designing both the grammar and the augments) from what is given in `jy.lisp` to include additional expression syntax beyond that provided in `G1`. You should provide the detailed productions and actions for logical expressions (and, or, not, `>, <`), functions of the form `f(), f(a), f(a,b)` etc; powers (right associative) of the form `a^b^c`; ordinary arithmetic, parenthesized expressions and the `if-then-else`. Optionally you may include other expressions if you wish.

Your guide for the code to be generated is fairly straightforward: *the augments should produce legal Lisp that could potentially be executed.* Demonstrate that your augmented grammar produces correct Lisp expressions by a sequence of inputs testing the major contributions of your extensions.

3. For your next use of the parser generator, you must come up with a complete grammar for MiniJava. This must pass through the LALR-generation process with hardly a peep. (We will allow if-then and if-then-else ambiguity.)

We are not telling you what the grammar is. You must figure it out from Appel's Appendix and the similarities to `G1` or other grammars you might have available to you. This can be either fairly easy or frustrating. Realize that you can always make a "smaller" language and grammar and then put such languages together.

To make this part of the project simpler, we will not require you to figure out anything particularly useful for augments. In fact we recommend that you use the default augment for every rule. This augment looks something like

```
(defun aug (&rest r) (if (null (cdr r)) (car r) r))
```

This function takes any number of arguments; if there is just one argument, it returns it. Goven more than one, it returns a list of them. We could have set up the parser generator to use a simpler default augment, such as

```
(defun aug (&rest r) r)
```

Why do you think we prefer the first augment?

Try parsing the examples of MiniJava programs you've written as well as the examples we've provided. Check them over for reasonableness with respect to precedence and associativity of operators, which you must specify through your grammar. That is, `a*b+c` must group the `a` and `b` under the `*`. Also `a*b*c` should group as `(a*b)*c` to conform to MiniJava semantics.

You should show that your parser correctly produces abstract syntax trees for enough examples to demonstrate all significant choices you've made. Do *not* give lengthy and duplicative examples; rather, just enough to show that you are doing the right thing. When we grade your project we will have a selection of other examples, including those from the `tests` files.

Assignment 4, preview: MiniJava "Abstract Syntax Tree"

For this assignment, your parser must translate the stream of tokens returned by the lexical analyzer into a Lisp s-expression described informally by the MiniJava Abstract Syntax, which is partially explained on pages leading up to figure 4.9 (page 99) of the text. In fact we won't use that exactly since there are some glitches in the text, and Lisp simplifies the construction and display of the AST compared to the class-based structure. The details of any particular construction can be viewed by seeing what our parser produces. It is possible that a discrepancy between your AST and ours is a mistake in our parser, in which case please bring it to our attention and we will try to see if "we meant to do that". Some of the ordering of declarations in the text's grammar may be relaxed in our parser; you may choose to follow us or the text.

In figure 1, we give a modified version of figure 4.9 we have come up with. Bring to our attention any case of discrepancies, please. Our table is similar to that given by Appel, but where Appel has

```
Foo(sometype1 somename1, sometype2 somename2, ...)
```

we will generally write

```
(Foo somename1 somename2 ...)
```

The terminals in this abstract grammar are id, num, true, false, and this, together with the types IntArrayType, IntType, and BooleanType. In order to take less room, we discard position information for everything but identifier nodes. When the names that appear in the grammar are plural, it indicates that a list; for example, a list of expressions has the form

```
;; Program structure
(Program MainClass ClassDecls)
(MainClass id id Statement)
(ClassDecl id (extends id | nil) VarDecls MethodDecls)
(VarDecl Type id)
(MethodDecl Type id Formals VarDecls Statements Exp)
(Formal Type id)

;; Allowed types
(Type base-type)
(IdentifierType id)

;; Allowed statements
(Block Statements)
(If Exp Statement Statement)
(While Exp Statement)
(Print Exp)
(Assign Identifier Exp)
(ArrayAssign Identifier Exp1 Exp2)

;; Allowed expressions
(And Exp Exp)
(Not Exp)
(LessThan Exp Exp)
(Plus Exp Exp)
(Minus Exp Exp)
(Times Exp Exp)
(IntegerLiteral int)
(BooleanLiteral true)
(BooleanLiteral false)
(ArrayLookup Exp Exp)
(ArrayLength Exp)
(Call Exp id Exps)
(IdentifierExp id)
(NewArray Exp)
(NewObject id)
this
```

Figure 1: Lisp abstract syntax tree, modified from Appel's figure 4.9

```
(Exps exp1 exp2 ...)
```

If there is a question about the translation, it is defined operationally by the provided complete parser. To use our parser together with `longerlex`, load `mj.fasl` and then `lex-adapt.fasl`. The source code for `lex-adapt` is also available in the `software/source` subdirectory. The main interface routine is `parse-java`, which takes a file name and returns the parse tree for the file. We also provide functions `print-java`, which pretty-prints a parse tree using Lisp's `pprint`; and `java-to-ast`, which reads a Java file and writes out a Lisp-readable file that stores the parse tree in a variable called `*ast*`. The MiniJava parser interface is also documented in the MiniJava interpreter documentation on the class web page.

Among the features of the abstract syntax form are

1. It can be printed on paper or to a file as ordinary Lisp data without loss of information (that is, a file can be read back in by Lisp).
2. As any data item, it can be "prettyprinted" in Lisp by, for example, `(pprint parsetree)` which will show it properly indented and more easily checked than otherwise.
3. The type-checker and interpreter can (and in future project parts will) use it as input. The interpreter may execute it. The type checker may produce further components of compilation, like assembly language, to be executed on a (virtual) machine.