

CS350 Lab 2: Parsing

Change the definition of `which_cs350_lab` in `tiger.cc` so that it has the value 2.

Modify the file `tiger-grammar.y` (and, if necessary, other files) to make a complete parser for tiger. This is discussed in the “programming exercise” section of Chapter 3 of Appel’s book (once again, ignore the discussion of specific files). Remember that grammars can not rule out every kind of error, and sometimes even if something could be ruled out in a grammar you may still choose to leave it for a later error detection pass. You should focus on two goals:

1. Accepting all legal tiger programs and rejecting many (but not all) illegal ones. For example, you should accept `1+2` and reject `3+else`; if you can reject `1+2:=a` with your grammar, that will save you a little work in Lab 3, but it is not worth suffering for. Your grammar does not have to reject illegal uses of `break`, distinguish *lvalues* from other expressions (e.g., `1+2:=a`), or distinguish type names (*type-id*) from other identifiers or expressions (your parser should accept `x [8] of 17` even if `x` is not the name of an array type; it can even allow `(3 + 4) [8] of 17`).
2. Successfully capturing the structure of the programs you accept. For example, the expression `if 1<2 then 3 else 4+5` should be equivalent to `if 1<2 then 3 else (4+5)`, *not* to `(if 1<2 then 3 else 4)+5`; if you accept `1+2:=a`, it should be structured as `(1+2):=a`, *not* `1+(2:=a)`.

You should try to minimize the number of shift-reduce and reduce-reduce conflicts in your grammar, through a combination of precedence and associativity directives, re-organizing your grammar rules, and attention to the Goals above.

For information about precedence and associativity, see Appel’s Grammar 3.35. Feel free to use standard tricks like the unary minus precedence (also from Appel’s Grammar 3.35) and the then/else precedence from the O’Reilly book on `lex & yacc` (by Levine, Mason, and Brown, and usually to be found in the CS lab — look up “if-then-else” in the index), but please cite things you get from books (in a comment) and do *not* just start experimenting with them in cases you don’t understand: you are likely to produce a conflict-free grammar that will not parse some inputs, or will parse them in ways that will cause trouble in the upcoming labs. My *strong recommendation* for knowing when to use precedence (and associativity) is this: when faced with a conflict (e.g., shift/reduce on `'|'` in state 73), try to come up with examples of legal expressions that would get you into that state (73) with a that symbol (`'|'`) as the next input. *After* checking for other problems (e.g., are there two different rules that both match these inputs, where you could have just one rule?), see if all these inputs involve one operator (or word) that should *always* have a higher precedence than the other (my example with “if 1<2” above came from such an investigation involving `if` and `+`). If this is so, feel free to use precedence (or associativity) rules.

If you cannot get down to zero conflicts, you may leave a small number of conflicts in your grammar, as long as you give a comment describing the source of each conflict and explaining why the default action will produce the appropriate result.

In addition to properly commenting your program and using clear names, you should record important design decisions in the file `Lab2-Parsing` in the `Design_Documents` subdirectory.

For 70% partial credit, do everything but arrays, records, and type declarations. This work will be sufficient to do the partial credit solutions for the later labs.