CS 3276 Project #3
Building a Parser

The purpose of this lab is to build a parser for the Tiger language. This parser is outlined on pages 82-83 of the class handout. The parser is to be built using **bison**, a tool for generating parsers.

A set of files is available to help you get started. The files are available on Brightspace in a compressed archive file `project3.tar.gz`. Download this file to your Linux system and extract the files from the archive using tar (use the command: `tar -xvf project3.tar.gz --gunzip`). The files in the archive include a Makefile, an initial `tig_parse.y`, and some auxiliary files. The files are C++ source files for the GNU g++ compiler. You should familiarize yourself with the contents of the files before attempting changes. Included are the following:

- `errormsg.cpp` & `errormsg.h` -- which aids in giving error messages with the correct line and character position
- `util.cpp` & `util.h` -- a small set of utility functions
- `parsetest.cpp` -- for testing the parser
- `Makefile` -- for compiling and linking all files
- `tig_parse.y` -- a bison grammar file for tiger; the file you will need to modify
- `lex.yy.c` -- a working lexical analyzer if you were not able to build one in Project #2
- `parsetest-good` -- a fully working executable that you can compare your parser against

In its present state, `tig_parse.y` parses files that contain only a single integer, a single **nil** expression, or the addition of two acceptable expressions (e.g., `1+2`). Your task is to extend `tig_parse.y` so that it parses all syntactically correct Tiger programs and rejects invalid Tiger programs. In creating a grammar for the Tiger language, you will likely encounter many shift-reduce and reduce-reduce conflicts. There are several ways of resolving such conflicts, including modifying the grammar or using precedence and associativity directives available in Yacc and Bison. In the end, your grammar should have as few shift-reduce conflicts as possible, and no reduce-reduce conflicts. For the basic assignment, you are not required to explain why any remaining shift-reduce conflicts are not harmful (as specified in the text). For this assignment, all embedded actions associated with grammar rules should be empty[1].

Extensions

In addition to the basic assignment, you may wish to do some or all of the following.

Extension 1. Tiger as presently defined treats semicolons as separators, not as terminators. There are a number of places where a C++ or Java programmer would expect to see a semicolon, that are illegal in Tiger, such as:

- `if a > b then c; else d`
- `let var x := 4; var y := 3; in ...`

Write your grammar in such a way that if an "illegal/superfluous" semicolon is encountered in some or all of these cases, that a warning message is given to the user, the semicolon is ignored, and the parse continues.

Extension 2. Perform syntax error recovery using Bison's "error" production. (See Section 3.5 of the handout). Note that if you attempt error recovery then you should produce an error/warning message that is appropriate for the error you are recovering from. You should not fix & accept wrong code without an error message.

[1] It is okay to have the embedded action call EM_error() if you attempt any of the extensions.

Extension 3. List all shift-reduce conflicts in your grammar (see the `y.output` file) and explain why each is not harmful, or use precedence and associativity directives to eliminate all conflicts.

Building and Testing

In order to successfully build your parser, you'll need to set things up properly. Once you have extracted the files from the archive, you should be able to go into the directory that contains the files (project3/) and type "make". It should run bison and then run the appropriate C++ compiler to build everything. If you decide to use your own lexical analyzer from project #2, you will need to copy your `tig_scan.l` file into your working directory and modify the `Makefile` to invoke **flex** (by un-commenting the appropriate commands). You will also need to change `tig_scan.l` so that it includes the file `y.tab.h` rather than `tokens.h`. The file `y.tab.h` is automatically generated by bison when it processes your `tig_parse.y` file.

At this point, you should be able to run the parser on a test file. To run it on a file `test.tig`, you would give the command

```
      ./parsetest test.tig
```

If your file contains nothing but a single integer, a single **nil** expression, or an addition expression (with integer, nil, or addition expressions as operands) [in addition to items ignored by the scanner (blanks, newlines, comments, etc.)] the parser should report that parsing completed successfully. For example, use the file `test2.tig` distributed with the project code. If the file contains anything else, it should give an error message stating the line number and character position of the first error.

As you improve `tig_parse.y`, the parser should recognize more and more program structures. I recommend that you add a few structures at a time and test. You will have to generate your own short test cases that contain only the structures your parser recognizes at that point. When your parser is completed, you can test it on the suite of Tiger test files previously supplied (available on Brightspace).

Project Submission

Submit your projects on Brightspace via the assignment page. You should submit the file `tig_parse.y` since that is the only file you should have modified. We will make an executable from scratch using your `tig_parse.y` file to make sure everything compiles cleanly. Also submit a `README` file that describes your project. The `README` file should state which parts and extensions of the project you completed, as well as describe any particularly interesting code or structures. Please also include an honor statement in the `README` file.

Grading

All students will start with a score of 90, a low-A grade (on a scale of 100). If you correctly implement the basic parser, 90 will be your grade. Your score can move up if you correctly implement any of the extensions. Your score can move down if your implementation is incorrect or has errors. Your score can also move down for poor programming style or the absence of a useful `README` file.

Hints

1. It is highly recommended that you add only a few grammar rules (2 or 3) at a time, resolving all shift/reduce and reduce/reduce conflicts before proceeding. Begin by adding additional expression rules.

For example, start by adding additional arithmetic rules that are similar to the existing grammar rule `exp -> exp PLUS exp`. Do not add the rules for Tiger declarations until needed (they are not needed until you get to the `let` expression). Note: you do not need to eliminate all shift/reduce conflicts, but you should investigate each to determine if the default shift action is the correct action. Add comments to your grammar file to explain why it is the correct action.

2. The tiger language is an *expression* language; that is, everything (besides declarations) returns a value. Thus it is possible to write some very strange looking programs; for example:
   ```
   let in end = if a then b
   ```
   (testing the equality of a `let` expression and an `if-then` expression). Such strange programs are syntactically correct. To insure that you parse such programs correctly, you need to keep all expressions as expressions; i.e., do not force your own hierarchical structure on the grammar such as creating statements, expressions, terms, factors, etc.
3. Use precedence and associativity directives to help resolve conflicts. See pages 154-155 and chapter 7 (pages 173-194) in the *flex & bison* book for a description of the directives and how to use them. See pages 74-75 in the handout for hints on handling unary minus. See pages 188-189 in the *flex & bison* book for hints on if-then-else conflicts.
4. Some tiger expressions are more like statements than expressions (e.g., the `while` expression). Some of these will cause conflicts. Use a trick similar to handling unary minus to give such expressions a low precedence.
5. Parentheses can appear in several different contexts as expressions. They appear in **sequences** (see page 522 of the Tiger Language Reference Manual), **no value** expressions (see page 522), and they can be used for syntactic grouping (see page 525). Thus it is valid for an expression to be a pair of parentheses containing zero, one, or two+ expressions. If two+, the expressions are semicolon separated. It is possible to generate a single expression grammar production that will handle all three cases. However, I recommend that you generate a separate grammar production for each, as that will prepare you for work in subsequent projects since we need to treat these three items differently.
6. To specify an epsilon move in the grammar, you simply leave the rule empty. It is good style to place a comment there; i.e., `/*epsilon*/`. You still need to place the empty braces for the semantic actions.
7. The array creation expression will lead to conflicts. Solve this by giving it a low precedence level. Also, due to the way that the tiger language is specified, you will need to use the following rule for array creation:
   ```
   lvalue [ exp ] of exp
   ```
   I.e., use an `lvalue` rather than an `id` in the rule. This modification to the grammar should remove the last shift/reduce conflict dealing with the array creation production. However, we will now need to add extra code to our type checker (in a future lab) to insure that whenever we encounter an array creation expression, that the first component of the expression is an ID and not some other type of lvalue.
8. Again, it is highly recommended that you add only a few grammar rules at a time. Test them before adding more rules (do not assume a clean compile means everything is okay). I state this twice since it always seems that a few students ignore this and end up wasting several hours of work.