CS 3276 Project #4
Building an AST

The purpose of this lab is to build an AST for Tiger programs. The structure of the AST is outlined in the class handout on pages 97-101. The AST is to be built via the semantic actions of your parser from Project #3.

A set of files is available to help you get started. The files are available on Brightspace in a compressed archive file `project4.tar.gz`. Download this file to your unix system and extract the files from the archive using tar (use the command: `tar -xvf project4.tar.gz --gunzip`). The files in the archive include a Makefile, an initial `tig_parse.y`, and some auxiliary files. You should familiarize yourself with their contents before attempting changes. Included are the following:

- `errormsg.[cpp|h]` -- which aids in giving error messages with the correct line and character position
- `util.[cpp|h]` -- a small set of utility functions
- `parsetest.cpp` -- for testing the parser and printing out the AST
- `parse.[cpp|h]` -- for calling your parser
- `Makefile` -- for compiling and linking all files
- `tig_parse.y` -- a bison grammar file for tiger; the file you will need to modify
- `absyn.[cpp|h]` -- AST declarations and constructor & printing functions
- `symbol.[cpp|h]` -- module to turn strings into symbols
- `table.[cpp|h]` -- a generic hash table to support a symbol table
- `lex.yy.c` -- a working lexical analyzer if you were not able to build one in Project #2
- `sample-outputs` -- a directory containing some output files you can compare your parser against

In its present state, `tig_parse.y` correctly parses all Tiger programs, but only builds ASTs for files that contain only a single identifier, a single integer, or a single string. Your task is to add semantic actions to all the rules in `tig_parse.y` so that it builds correct ASTs for all valid Tiger programs. The nodes that should be built for each construct are largely described on pages 97-101 of the handout. Please read these pages carefully. But please note that we've replaced Appel's old C-style constructs with a true object-oriented C++ solution. For every constructor function mentioned in the Appel document, you should be able to find a corresponding object constructor in the provided OOP code. You should also note (from Appel) that certain language constructs do not have explicit AST-nodes, but are converted to equivalent expressions:

- The expression `a & b` yields an AST equivalent to `if a then b else 0`.
- The expression `a | b` yields an AST equivalent to `if a then 1 else b`.
- The expression `-a` yields an AST equivalent to `0-a`.

Note that it is okay to use a null pointer as a parameter **only** in the following cases:

- any "`list`" parameter
- the "`else`" statement in an `if-then-else`
- the "`typ`" parameter when creating an `A_VarDec`
- the "`result`" parameter when creating an `A_Fundec`

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 (project4/) and type "make". It should run bison (two shift/reduce conflicts will be reported) and then run the C++ compiler to build everything. I recommend that you use my `tig_parse.y` file rather than using the file you created for project #3 -- this will ensure that our parsers all work identically and will ease the task of finding bugs. 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.

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 parser correctly builds an AST for the program being parsed, the AST will be printed to the screen[1]. Be sure to compare the output of your parser against the few sample files in the `sample-outputs` directory.

I recommend that you add code for 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 handles at that point. When your parser is completed, you can test it on the files Dr. Appel has supplied (also available on Brightspace). I have also supplied a test script that runs your parser on all the Appel test cases.

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 of the project you completed and which were not completed, as well as describe any particularly interesting code or structures.

Grading

There are no extensions to this lab. If you correctly implement the lab, your grade will be 100. 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 <u>highly</u> recommended that you add code for only a few grammar rules (2 or 3) at a time.
2. Take some time to look through the `tig_parse.y` file before you begin your modifications. Also take a glance at all the other source files provided -- you just might learn something.
3. Make sure you understand the types associated with the semantic values of grammar symbols. These types have already been specified for you in the `tig_parse.y` file (see the `%type` specifications). Understanding these types will help you understand what you are receiving when the RHS of a production is reduced and what is expected to be produced for the LHS symbol. In the past, this has been the biggest stumbling block for students. Once they understood the relationship of grammar symbols to types then their progress on the lab went fairly smoothly.
4. Use the S_Symbol() constructor to create s_symbol objects, and not S_symbol(). This will be very important to set you up for the next assignment.

[1] Please note that the parsetest.cpp file reports *"Parse successful"* if a non-null root pointer is returned by the parser. It reports *"Parse failed"* if the root pointer is null. Thus success/failure is determined by receiving a root pointer and not whether the grammar is correct or not (or whether the AST is correct or not).