Project 1: A scanner and parser

Your combined project for this semester is to implement a compiler from a pedagogical language, Tiger, to the MIPS instruction set. Tiger is:

- *imperative*: a Tiger program executes by executing statements in sequence, each of which updates persistent program state. This is similar to, e.g., C and Java, but dissimilar to functional languages such as Scheme and Haskell or logical languages such as Prolog.
- *procedural*: a Tiger program may contain multiple procedures, similar to basically all modern languages.
- *first-order*: as a Tiger program executes, it only creates new numeric values and arrays of numeric values, not new functions. This is similar to, e.g., C and Java and dissimilar to, e.g., Scheme and Haskell.
- *statically-typed*: each Tiger value belongs to a type, which is specified in the program. Valid Tiger programs only perform operations on values at runtime that are consistent with the types given to the values statically.

In other words, Tiger is basically a simple subset of C and Java.

The entire compiler will be implemented as a sequence of phases:

1. A scanner.
2. A parser.
3. Semantic analyses.
4. A generator for intermediate representation.
5. A MIPS code generator, which will perform instruction selection and register allocation.

The first project is to implement the first two phases: a scanner and parser.

1 Phase 1: a scanner

The scanner phase of your compiler should define a space of tokens, each of which corresponds to a lexeme defined in the Tiger language specification. At runtime, you scanner should take a given input program and generate the longest sequence of tokens that, combined, match a prefix of the input program. This interface differs slightly from the interface supported by a scanner described in class, which technically only requires the next token of input. The difference will allow you to evaluate the correctness of your scanner independent of the tokens requested by the parser that you implement in Phase 2.

You have complete freedom in designing the scanner: your grade concerning the scanner will be completely determined by the sequence of tokens that it outputs. One reasonable design to follow would be to structure the scanner as executing in two steps: an initialization step that is performed before the scanner performs any operations on an input stream and an actual matching step. In the initialization step, the scanner would construct a map from each lexical token T to a deterministic finite automaton (DFA) that recognizes T. To construct each DFA, the scanner would take a regular expression that defines the language of each token T and construct a DFA D_T that recognizes T by first constructing an NFA N_T that recognizes T (using Thompson's construction) and then constructing a D_T as a DFA that recognizes the same language as N_T (using the subset construction).

In the matching step, the scanner would read characters as input from an opened input stream and simultaneously run the DFA for each token on the input characters read. When the scanner determines that each DFA has reached a dead state, it would push back onto the stream all read characters until the remaining unpushed characters are the longest string matched by the DFA for some token T. The scanner would then return T.

There are several strings that belong to the languages of multiple tokens. In each case, the string belongs to both a keyword class and the class `id`. When the scanner reads such a string s, it should always recognize s as a *keyword*.

2 Phase 2: a parser

The parsing phase of your compiler should interact with the scanner to construct the syntax tree of an input program, determined by the grammar given in the language reference. Given an input program P, (1) if P is a syntactically valid Tiger program, then the parser should generate the parse tree of P (defined in §3); (2) if P is not a syntactically valid Tiger program, then the parser should output a suitable error message (defined in §3).

You have a large degree of freedom in designing the parser, as long as it generates the correct parse tree of an input program. One reasonable design of the parser would be the following.

1. From the Tiger grammar G given in the language specification, generate an equivalent grammar G' that can be parsed by an $LL(1)$ parser by removing left recursion and left-factoring the result. G' can be generated by a combination of manual and automatic work.

2. Generate the $LL(1)$ parsing table of G'. The table can be generated by a combination of manual and automatic work.

3. Run the (fixed) $LL(1)$ parsing algorithm using the parsing table generated to construct a syntax tree T' for the input program P.

4. Apply a sequence of transformations to T' to transform it to a parse tree of P under G. These transformations will remove parse rules introduced in the process of removing left recursion and left-factoring.

3 Evaluation

Please turn in the complete source code of your project in a zipped tarball (i.e., a package created by running `tar czf`). Include in your tarball a Makefile such that the command `make` generates a jar file `parser.jar` containing the complete implementation of your parser. Your implementation should support two optional flags:

- If the parser is given flag `--tokens`, then it should output to standard output the longest sequence of valid tokens that it encounters before an error. In particular, let the *string representation* of each lexeme s be defined as follows. If s a member of one of the keyword lexemes T, then the string representation of T is the literal name of T. If s is a member of one of the class lexemes T, then the string representation of s is s followed by the literal name of T, separated by a colon. E.g., the string representation of `0124` is `0124:intlit`. The compiler should output the sequence of string representations of each token matched from the input.

- If the parser is given flag `--ast` and is given a valid Tiger program P, then it should output the S-expression of P to standard output. The *S-expression* of a syntax tree T, denoted $\mathsf{sexpr}(T)$, is a string representation of T, defined as follows. **(1)** If T is constructed from only a terminal `T`, then the S-expression of T is simply the string representation of the string as a member of T. **(2)** for each non-terminal A and grammar symbols $a_0, \ldots, a_n$, if T is constructed by applying parse rule $\mathrm{A} \to a_0 a_1 \ldots a_n$ to syntax trees $T_0, T_1, \ldots, T_n$, then $\mathsf{sexpr}(T) =$ `A (`$\mathsf{sexpr}(T_0)$`)(`$\mathsf{sexpr}(T_1)$`) ... (`$\mathsf{sexpr}(T_n)$`)`. If the parser is given both flags `--tokens` and `--ast`, then it should output the string of tokens, followed by the S-expression of the parse tree.

Independent of whether or not your compiler is run with flags `--tokens` or `--ast`, if it is given an input that is not a syntactically valid Tiger program, then it should output a suitable error message to standard error. In particular, if it cannot match the entire input as a sequence of tokens, then it should output the line number of character position from

which it could not match any further tokens. If it was able to match the entire input as a sequence of tokens but could not parse the input, then it should output to standard error the string representation of the token that it read to determine that it could not parse the input.

We will run your parser on a program (stored in filename, say, `program.tgr`) by running the command

```
java -jar parser.jar program.tgr
```

(with optional flags `--tokens` and `--ast`)

You have complete freedom in designing how your compiler is built, so long as the command `make` generates a suitable jar file `parser.jar`. In particular, you can use another build system to perform all of the actual build work and include a Makefile in which the target `all` simply runs the actual build command.

To evaluate your parser, we will run it on a set of test Tiger programs and compare the result to a reference answer. To partially automate grading, we will by default evaluate the compiler as correct on an input program only if the result is identical to the reference answer, ignoring whitespace. Given that the Tiger grammar is unambiguous and the S-expression of a syntax tree is determined by the tree, for any given Tiger program, there should be a single correct output. However, we will inspect all output flagged automatically as incorrect to confirm that the output is actually incorrect and to award partial credit on a case-by-case basis.

Points will be awarded in the following categories:

Component	Points Possible
Scanner	40
Parser	60
Total	100

We may, at a future date, provide a reference implementation of a parser and a selection of test programs.

4 Discussion

We will award identical grades to each member of a given project team, unless members of the team directly register a formal complaint. We assume that the work submitted by each team is their work solely. Any non-obvious discussion or questions about design and implementation should be either posted on the course's Piazza message boards privately for the instructors or presented in person during office hours or after lecture. If the instructors determine that parts of the discussion are appropriate for the entire class, then they will forward selections. Under no condition is it acceptable to use code written by another team. The instructors will automatically check each submitted solution against other solutions submitted and against other known implementations.