Progettino Lexer

1 Overview

I progettini Lexer e Parser vi porteranno a progettare e costruire parte del front-end di un compilatore per il linguaggio di programmazione che vi è stato assegnato. I due progettini copriranno le prime due componenti del compilatore: l'analisi lessicale e quella sintattica. Il Lexer dovrà interfacciarsi al Parser come descritto a lezione. I progettini andranno svolti in Java.

Per quanto riguarda il Lexer, dovrete produrre un analizzatore lessicale utilizzando il *generatore di analizzatore lessicale* JFlex. Basandovi sulla specifica lessicale del linguaggio, estratta dal manuale di riferimento che vi sarete procurati, dovrete creare un file .jflex contenente tutte le definizioni necessarie a JFlex per generare in modo automatico il codice Java del Lexer.

Questo progettino va sviluppato da solo o in coppia.

2 Materiale da consultare

La documentazione e tools a cui fare riferimento include:

- jflex
  Manuale e software jflex e, se si intende lavorare su Eclipse, il plugin Cup-Lex-Eclipse. (Riferimenti ad essi sono in allegato)
- Esempio completo Pascal
  Un esempio compilabile ed eseguibile di un Lexer per il linguaggio Pascal. (In allegato è data una cartella con i files necessari ed un README).
- Esempio Cool
  Un esempio di specifica jflex per il linguaggio Cool e sua specifica lessicale. (I due files sono in allegato)
- Manuale di riferimento del linguaggio
  Manuale che include sia la specifica lessicale che sintattica (grammatica) del linguaggio che si vuole implementare. La specifica sintattica sarà utile per decidere quali tokens riconoscere. (Il manuale di riferimento è quello che dovete inserire, o che avete già inserito, nel wiki del sito del corso).
- Compilatore/interprete "ufficiale" del linguaggio
  Va scaricato direttamente dal sito web del linguaggio e funge da completamento del manuale di riferimento. In linea teorica stiamo cercando di "riprodurne" la fase lessicale e sintattica.

Si consiglia di comprendere profondamente il funzionamento dell'esempio Pascal e l'esempio Cool prima ancora di continuare la lettura di questo documento.

3 Input/output dell'analizzatore lessicale

Il vostro analizzatore lessicale dovrà essere in grado di prendere, come input, qualsiasi sequenza di parole e, per ciascuna di esse, riconoscerne l'eventuale appartenenza al linguaggio. Nel caso essa appartenga, essa va classificata associandole un token con eventuale attributo ed eventuale inserimento nella tabella dei

simboli. Nel caso non appartenga, va restituito al parser un token ERROR (come visto nell'esempio Cool) il cui attributo è dato da un messaggio di errore coerente con quello che darebbe il compilatore/interprete originale e/o con le specifiche del linguaggio .

L'analizzatore dovrà cercare di individuare tutti i possibili tokens senza arrestarsi al primo errore.

Per poter testare il vostro analizzatore lessicale dovrete costruire dei files di test, una strategia di testing ed un programma java che, per chiamate successive, sia in grado di ricevere i tokens dal lexer per poi stamparli insieme alla loro posizione nel file (si veda, al riguardo, l'esempio Pascal).

È importante ricordare che le azioni del lexer non devono mai stampare, ma possono solo restituire tokens (come nell'esempio Cool).

4 Consegna

Il software prodotto deve essere consegnato come un unico .zip sul sito del corso.

Il file zip sottomesso deve contenere i seguenti files:

- *Relazione finale.rtf*: file contenente le proprie annotazioni sul progetto (seguire lo schema proposto in allegato);
- *good_1.test, . . . , good_n.test*: files di test prodotti contenenti codice corretto;
- *bad_1.test, . . . , bad_n.test*: files di test prodotti contenenti codice non valido;
- *nomelinguaggio.lex*: contenente la specifica jflex dell'analizzatore lessicale;
- *programma java di test*: contenente il main che richiama l'analizzatore lessicale;
- *classe/i di supporto*: quali ad esempio quelle per la gestione della tabella dei simboli.

Il software prodotto dovrebbe generare un output il più coerente possibile con il compilatore/interprete ufficiale.

5 Valutazione

Il progettino verrà valutato in base alla corrispondenza con le specifiche ma anche in base alla qualità della relazione finale e dei test cases.

6 Note ulteriori

Le seguenti sottosezioni riguardano considerazioni valide per i linguaggi di programmazione in generale e quindi possono o non possono essere utili ai fini del vostro specifico progetto. Ad esempio, vengono specificati i testi dei messaggi di errore che, nel nostro caso, devono comunque essere conformi ai messaggi del nostro specifico compilatore/interprete.

Si consiglia comunque di prenderne nota.

6.1 Scanner Results

Your scanner should be robust—it should work for any conceivable input. For example, you must handle errors such as an EOF occurring in the middle of a string or comment, as well as string constants that are too long. These are just some of the errors that can occur; see the manual for the rest.

You must make some provision for graceful termination if a fatal error occurs. Core dumps or uncaught exceptions are unacceptable.

6.1.1 Error Handling

All errors should be passed along to the parser. You lexer should not print anything. Errors are communicated to the parser by returning a special error token called **ERROR**. There are several requirements for reporting and recovering from lexical errors:

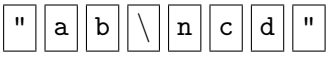
- When an invalid character (one that can't begin any token) is encountered, a string containing just that character should be returned as the error string. Resume lexing at the following character.
- If a string contains an unescaped newline, report that error as `‘‘Unterminated string constant’’` and resume lexing at the beginning of the next line—we assume the programmer simply forgot the close-quote.
- When a string is too long, report the error as `‘‘String constant too long’’` in the error string in the **ERROR** token. If the string contains invalid characters (i.e., the null character), report this as `‘‘String contains null character’’`. In either case, lexing should resume after the end of the string. The end of the string is defined as either
  1. the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
  2. after the closing **"** otherwise.
- If a comment remains open when EOF is encountered, report this error with the message `‘‘EOF in comment’’`. Do *not* tokenize the comment's contents simply because the terminator is missing. Similarly for strings, if an EOF is encountered before the close-quote, report this error as `‘‘EOF in string constant’’`.
- If you see an end-of-comment marker outside a comment, report this error as `‘‘Unmatched <end-of-comment marker>)’’`, rather than tokenzing its components.

6.1.2 String Table

Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table*.

subsubsectionStrings

Your scanner should convert escape characters in string constants to their correct values. For example, if the programmer types these eight characters:



your scanner would return the token **STR_CONST** whose semantic value is these 5 characters:

where \n represents the literal ASCII character for newline.

6.2 Java Notes

Each call on the scanner returns the next token and lexeme from the input. The value returned by the method **CoolLexer.next_token** is an object of class **java_cup.runtime.Symbol**. This object has a field representing the syntactic category of a token (e.g., integer literal, semicolon, the `if` keyword, etc.). The syntactic codes for all tokens are defined in a file usually produced by JavaCup. The component, the semantic value or lexeme (if any), is also placed in a **java_cup.runtime.Symbol** object. The documentation for the class **java_cup.runtime.Symbol** as well as other supporting code is available at

`http://czt.sourceforge.net/dev/java-cup-runtime/apidocs/java_cup/runtime/Symbol.html`