

## CS 602/702 COMPILER DESIGN I PROJECT

*Due Thursday, May 5, 2005, at 5:00 P.M.*

Implement language description and implementation tools based on Two-Level Grammar (TLG), a specification language for programming language syntax and semantics. This will effectively be a compiler for TLG.

1. The lexical analyzer should recognize all necessary tokens in the TLG source program. The primary tokens are:
  - *Meta-symbols.* `: :: , ; . | ( ) { } [ ] + *`
  - *Operators.* `+ - * / || < <= > >= = != not`
  - *Keywords.* Keywords begin with a lower case letter and consist only of lower case letters.
  - *Identifiers.* Identifiers begin with an upper case letter and then may be followed by zero or more letters, digits and underscores such that there are no two consecutive underscores and an underscore does not appear at the end. An identifier may followed by the words `List` or `Set` (e.g. `StudentList`).
  - *Constants.* There are six types of constants: 1) Integer - signed strings of one or more decimal digits, 2) Float - signed strings of one or more decimal digits with a decimal point and possibly an exponential qualifier (e.g. `1.2E+6`), 3) Boolean - the strings `true` and `false`, 4) String - strings of characters, possibly including the special string `Empty`, optionally enclosed in double quotes, and 5) Character - a string of length one, enclosed in single quotes.

The TLG specification text should also be output in this phase.

2. The parser should recognize exactly the described syntax. If there are any syntactic errors, an appropriate error message should be output and compilation may be terminated. A concrete syntax of the subject language expressed in extended BNF is given on the last page. Note that `{`, `}`, `[`, and `]` are terminal symbols as well as grammar symbols, the former distinguished from the latter by being in typewriter font. The start symbol of this grammar is `specification`.
3. Types include integers, floating points, Booleans, strings, characters, first-class objects, lists, sets and maps of these types, and functions (methods) returning values of these types. Type checking should make sure that these data types are used correctly. All functions are public and all variable identifiers are private.
4. The symbol table should contain all identifiers used in the specification along with the appropriate attributes. The symbol table may be considered as an environment in which static semantics checking can be done.
5. There will be three types of target code generated. One will be a lexical specification suitable for input to a lexical analyzer generation tool such as JLex. Another will be a syntax specification suitable for input to a parser generation tool such as CUP. The third will be an executable program to interpret semantics in a high-level language such as C++ or Java.

The compiler may be written in any programming language that you wish and implemented on any computer that you wish. In the past, most projects have benefited from the Java software tools JLex and CUP which allow automatic generation of lexical and syntactic analyzers. These

tools expect to interface with programs written in Java. There are similar tools available for C and C++. It is also possible to use alternative languages by adding additional passes in the compiler. For example, CUP could be used to generate a tree representation of the program which could be output in a form compatible with programming language L. Programs in L could then read this intermediate form to complete the additional compilation phases.

It is suggested that you keep the following tentative schedule in order to guarantee your successful completion of this project. The schedule may be revised based on class coverage of the necessary prerequisite lecture material. Furthermore, the phases of the project listed below may be further decomposed for purposes of better modularity.

<u>Compiler Phase</u>	<u>Estimated Completion Date</u>
Lexical analyzer	Wednesday, January 19
Syntax analyzer	Monday, February 14
Symbol table construction	Monday, February 28
Type checking routines	Monday, March 21
Code generation	Thursday, May 5

## Two-Level Grammar Syntax

specification	::=	{interface-definition} class-definition {class-definition}
interface-definition	::=	<b>interface</b> class-identifier [ <b>extends</b> class-identifier {, class-identifier}] . {interface-member-declaration} <b>end interface</b> [class-identifier] .
interface-member-declaration	::=	declarator   function-signature
class-definition	::=	<b>class</b> class-identifier [ <b>extends</b> class-identifier {, class-identifier}] . {class-member-declaration} <b>end class</b> [class-identifier] .
class-member-declaration	::=	declarator   function-definition
declarator	::=	type-identifier {, type-identifier} :: type-alternatives .
type-alternatives	::=	type {; type}
type	::=	type-string {type-string}   <b>map</b> type-identifier <b>to</b> type-identifier
type-string	::=	identifier-string   [ identifier-string ]   { identifier-string } repeated
identifier-string	::=	type-identifier   keyword   constant   binary-operator   unary-operator   meta-symbols
repeated	::=	*   +
function-definition	::=	function-signature .   function-signature : statement-list {; statement-list} .
function-signature	::=	string {function-string}
function-string	::=	string   keyword   type-identifier
statement-list	::=	statement {, statement}
statement	::=	assignment-statement   conditional-statement   set-statement   update-statement   expression
assignment-statement	::=	type-identifier := expression
conditional-statement	::=	<b>if</b> expression , <b>then</b> statement-list , [ <b>else</b> statement-list ,] <b>end if</b>
set-statement	::=	class-identifier <b>set</b> type-identifier <b>to</b> expression
update-statement	::=	<b>update</b> map-identifier <b>with</b> type-identifier <b>mapped to</b> expression
expression	::=	term   expression binary-operator expression
term	::=	primary-expression   unary-operator term
primary-expression	::=	identifier-string {identifier-string}   ( expression )   <b>apply</b> -expression   <b>get</b> -expression
apply-expression	::=	<b>apply</b> map-identifier <b>to</b> expression
get-expression	::=	class-identifier <b>get</b> type-identifier

**Syntactic and Semantic Conventions** The keywords and the token symbols are in typewriter font. The unary operators are +, − and **not** (logical negation). Binary operators obey the customary precedence rules, from highest to lowest:

multiplicative	*, /
binary additive	+, −,
relational	<, >, <=, >=, =, !=

class-identifiers, type-identifiers, and map-identifiers are the same lexical/syntactic items as identifiers but have the semantics given by the appropriate qualifier. A string is any sequence of characters not beginning with an upper-case letter (i.e. not an identifier but possibly a keyword - no reserved words) and not containing spaces (e.g. `syntax`, `end`, `true`, `+`, `602`, `3.14`, etc.). Strings may optionally be enclosed in double quotes (`""`). Characters are single characters enclosed in single quotes (`'`). Note that there are no reserved words in TLG, but you may assume that if the first string of a statement or expression is a keyword, then it may be interpreted as that keyword. In lexical analysis, it may be convenient to separate token strings of letters from integer, floating point, and Boolean constants (`true` and `false`).