CS 4240: Compilers and Interpreters
Project Phase 1: Scanner and Parser
Due Date: October 4th 2015 (11:59 pm) (via T-square)

Introduction

This semester, through a project split into 3 phases, we are going to build a full compiler for a small language called Tiger targeted towards the MIPS machine. The phases of the project will be:

Phase I (this phase): To build a scanner and parser for Tiger language.
Phase II: Semantic Analysis and intermediate representation (IR) code generation
Phase III: Instruction selection, register allocation and MIPS code generation

The purpose of this phase is to build a scanner and parser for the Tiger language using a table driven scanner and a table driven LL parser; both scanner and parser tables will be hardcoded for this language and not generated by a tool.

Please use the attached language specifications for Tiger given. It is a small language with properties that you are familiar with: functions, arrays, integer and float types, control flow, etc. – the syntax and semantics of the Tiger's language constructs are described in the document along with a sample program.

Phase I Description

First you will build a scanner that will scan the input file containing the Tiger program and perform lexical actions and return tokens one by one on demand to the parser.

Part 1: DFA

First, develop a DFA for your scanner. The DFA will be implemented through a table which will show state change based on input character. It should transition based on the lexical rules, reserved words, and punctuation symbols of Tiger, as described in the language document. The DFA should enter an accept state as soon as it can make a decision about the current token where the token is generated and sent to the parser.

The following token types are possible in Tiger:

COMMA COLON SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE PERIOD
PLUS MINUS MULT DIV EQ NEQ LESSER GREATER LESSEREQ GREATEREQ AND OR ASSIGN

Keyword tokens: ARRAY BREAK DO ELSE END FOR FUNC IF IN LET NIL OF THEN TO TYPE VAR WHILE ENDIF BEGIN END ENDDO

ID INTLIT FLOATLIT

(The first block includes all punctuation. The second block includes all keywords. The third block includes user-defined identifiers, integer and float literals.)

Notes:

- The keywords are recognized as a subset of the identifiers – that is first you will recognize an identifier (ID) and then check the string against a list of keywords, if it matches you return a keyword token and not an ID.
- Use the longest match algorithm when recognizing tokens. That is you will keep matching the input character to the current token, until you encounter one which is not a part of the current token.
- Write one big DFA by first writing small DFAs for each of the tokens and then combining them through NFA and then convert the NFA to DFA. Since most of the tokens have different prefixes, the NFA to DFA conversion should be straightforward.

Part 2: Scanner

Write a table-driven scanner for Tiger using the above DFA table. The scanner will read in char by char from the input file and perform state transition based on the DFA table and at the end generate the token in its final state. Thus, it should be able to read in a stream of characters and, on cue, return the next matched <token type, token> tuple, or it should throw an error. For example, given the stream "var x := 1 + 1", the first request to the scanner for a matching token should return <VAR, "var">, the second call <ID, "x">, and so forth.

Write the scanner in Java without using any regex class – scanner should just rely on the DFA table you will supply to it. Its behavior for programs that conform to Tiger's lexical requirements should be a graceful: It should be able to read in the program's stream of characters and return the correct token tuple on each request. For lexically malformed Tiger programs, the scanner should throw an error which prints: line number in the file – the partial prefix of the erronous string (from the input file), the malformed token putting it in quotes pin-pointing the culprit character that caused the error. The scanner should be capable of catching multiple errors in one pass – ie, should not quit but continue on after catching the 1st error. It will throw away the bad characters and restart the token generation from the next one.

Part 3: Parser

Write an LL(1) parser for Tiger. This consists of three parts:

1. Rewrite the grammar given in the Tiger language specification below to remove the ambiguity by enforcing operator precedences and left associativity.
2. Modifying the grammar obtained in step 1 to support LL(1) parsing. This could include removing left recursion and performing left factoring on the grammar obtained in step 1

above. This part is to be done by hand.

3. Creating the LL(1) parser table for Tiger. This will drive the decision-making process for the parser. This part is to be done by hand.
4. Writing the code for the parser which uses stack and performs LL(1) parsing by predicting the right rule for expansion to perform incoming token matching.

Write the parser in Java. For syntactically correct Tiger programs, the parser should output "successful parse" to stdout. For programs with lexical issues, the scanner is already responsible for throwing an error. For programs with syntactic problems, however, the parser is responsible for raising its own errors. In these cases, the output should be some reasonable message about the error, which should include : input file line number where it occurred, a partial sentence which is a prefix of the error, the erroneous token and perhaps what the parser was expecting there. In addition, your parser should also output the sequence of token types it sees, as it receives them from the scanner when you turn on a debug flag in your code. This will help us in verifying your solution. For example, given the stream "var x := 1 + 1", the parser would output "VAR ID ASSIGN INTLIT PLUS INTLIT". You will implement simple error recovery mechanism in the parser. You will try to delete the bad token and continue the parse, if the next token is also bad you will repeat the process. In worst case you will delete all the token until you encounter semi-colon (that terminates the sentence) and then continue the parse with the next sentence. More advanced error recovery mechanisms than the above are not expected to be built and are out of scope of this project.

Part 4: Turn-in

Correctness

You will be provided with some simple programs for testing. You will also be provided with at least one large program for which you will have to match the final output.

Recommended: It is strongly recommended although not mandatory to get your DFAs, LL(1) grammar modifications, and parsing tables (all hand-written parts) checked by the TA as you work towards their implementation. This might save the debugging cycle. Each team can submit this as a formal document to the TA who will check and give feedback to the team. No grade will be assigned during this process of giving feedback (the teams are expected to correct the problems cited by the TA – he will not give solutions).

Grading

Deliverables for phase I

1.	Hand-written DFA for the scanner (could be in the form of the table)	(10 points)
2.	Scanner code	(30 points)
3.	Hand-modified Tiger grammar in appropriate LL(1) grammar form	(10 points)
4.	Hand-written parser table	(15 points)
5.	Parser code	(30 points)
6.	Testing and output report	(5 points)

Tiger Language Reference Manual

Credit: Modified from Stephen A. Edwards' "Tiger Language Reference Manual" and from Appel's Modern Compiler Implementation in C

Grammar

<tiger-program> → let <declaration-segment> in <stat-seq> end

<declaration-segment> → <type-declaration-list> <var-declaration-list> <funct-declaration-list>

<type-declaration-list> → NULL
<type-declaration-list> → <type-declaration> <type-declaration-list>

<var-declaration-list> → NULL
<var-declaration-list> → <var-declaration> <var-declaration-list>

<funct-declaration-list> → NULL
<funct-declaration-list> → <funct-declaration> <funct-declaration-list>

<type-declaration> → `type` <id> = <type> ;

<type> → <type-id>
<type> → `array [INTLIT] of` <type-id >

<type-id> → int | float

<var-declaration> → var <id-list> : <type-id> <optional-init> ;
<id-list> → id
<id-list> → id, <id-list>
<optional-init> → NULL
<optional-init> → := <const>

<funct-declaration> → `function` id (<param-list>) <ret-type> begin <stat-seq> end ;
<param-list> → NULL
<param-list> → <param> <param-list-tail>
<param-list-tail> → NULL
<param-list-tail> → , <param> <param-list-tail>
<ret-type> → NULL

<ret-type> → : <type-id>
<param> → id : <type-id>

<stat-seq> → <stat>
<stat-seq> → <stat> <stat-seq>

<stat> → <lvalue> := <expr> ;
<stat> → `if` <expr> `then` <stat-seq> `endif ;`
<stat> → `if` <expr > `then` <stat-seq> `else` <stat-seq> `endif;`
<stat> → `while` <expr> `do` <stat-seq> enddo;
<stat> → `for` id := <expr> `to` <expr> `do` <stat-seq> enddo;
<stat> → <opt-prefix> id(<expr-list>) ;
<opt-prefix> → <lvalue> :=
<opt-prefix> → NULL
<stat> → break;
<stat> → return<expr> ;

<expr> → <const>
→ <lvalue>
→ <expr> <binary-operator> <expr>
→ (<expr>)
<const> → INTLIT
<const> → FLOATLIT

<binary-operator> → + | - | * | / | = | <> | < | > | <= | >=| & | |

<expr-list> → NULL
<expr-list> → <expr> <expr-list-tail>
<expr-list-tail> → , <expr> <expr-list-tail>
<expr-list-tail> → NULL

<lvalue> → id <lvalue-tail>
<lvalue-tail> → [<expr>]
<lvalue-tail> → NULL

Lexical Rules

Identifier: sequence of one or more letters, digits, and underscores. Must start with a letter can be followed by zero or more of letter, digit or underscore. Case sensitive.
Comment: begins with /* and ends with */. Nesting is not allowed.
Integer constant: sequence of one or more digits. Should not have leading zeroes. Should be unsigned.

float constant: Must have at least one or more digits before the decimal points followed by a decimal point. There could be zero or more digits after the decimal point. Floats are also unsigned.

Reserved (key) words

`array break do else end for function if in let nil of then to type var while endif begin end enddo return`

Punctuation Symbols

, : ; ( ) [ ] { }

Binary Operators

+ - * / = <> < > <= >= & |

Assignment operator

:=

Precedence (Highest to Lowest)

() * / + - = <> > < >= <= & |

Operators

Binary operators take integer operands and return an integer result. Inequality operators compare their operands, which may be either both integer or both float and produce the integer 1 if the comparison holds and 0 otherwise. The binary operators = and <> can compare any two operands of the same type and return either integer 0 or 1. Integers are the same if they have the same value. The binary operators +, -, *, and / require two operands and return result (mixed expression between integers and floats is permitted). The +, -, *, and / operators are left associative.

Zero is considered false; everything else is considered true. Parentheses group expressions in the usual way. The comparison operators do not associate, e.g., a=b=c is erroneous. The logical operators & and | are logical "and" and "or" operators. They take a logical and or or of the conditional results and produce the combined result. String comparison is done using normal ASCII lexicographic order. Any aggregate operation on arrays is illegal – they must be operated on an element by element basis.

Arrays

An array of any named type may be made by `array [intlit] of` *type-id* of length intlit. Arrays can be created only by first creating a type. Array of array are supported in a similar manner by first creating a base type for the base array and then can be used to create an array from it.

Types

Two named types `int` and `float` are predefined. Additional named types may be defined or redefined (including the predefined ones) by type declarations.

The two production rules for *type* (see the grammar rules above) refer to

1. a type (creates an alias in a declaration)
2. an array from a base type

Type equivalence enforced in the compiler is name type equivalence – ie, variables a and b defined to be of same named type are considered equivalent. In other words, even if two entities are structurally equivalent (such as two arrays of same length of integers), they will not be treated equivalent by the compiler.

In `let` . . . *type-declaration* . . . `in` *expr-seq?* `end`, the scope of the type declaration begins at the start of the sequence of type declarations to which it belongs (which may be a singleton) and ends at the end. Type names have their own name space.

Assignment

The assignment expression *lvalue* := *expr* evaluates the expression then binds its value to the contents of the *lvalue*. Assignment expressions do not produce values, so something like `a := b := 1` is illegal.

Control Flow

The if-then-else expression, written `if` *expr* `then` *<stat-seq>* `else` *<stat-seq>* `endif` evaluates the first expression, which must return an integer. If the result is non-zero, the statements under the then clause are evaluated, otherwise the third part under else clause is evaluated.

The if-then expression, `if` *expr* `then` *<stat-seq>* `endif` evaluates its first expression, which must be an integer. If the result is non-zero, it evaluates the statements under then clause.

The while-do expression, `while` *expr* `do` *<stat-seq>* evaluates its first expression, which must return an integer. If it is non-zero, the body of the loop <stat-seq> evaluated, and the while-do expression is evaluated again.

The for expression, `for` *id* := *expr* `to` *expr* `do` *<stat-seq>* evaluates the first and second expressions, which are loop bounds. Then, for each integer value between the values of these two expressions (inclusive), the third part <stat-seq> is evaluated with the integer variable named by *id* bound to the loop index. This part is not executed if the loop's upper bound is less than the lower bound.

Let

The expression `let` *declaration-list* `in` *<stat-seq>* `end` evaluates the declarations, binding types, variables, and functions to the scope of the expression sequence, which is a sequence of zero or more semicolon-separated statements in <stat-seq>.

Variables

A *variable-declaration* declares a new variable and its initial value (optional).. A variable lasts throughout its scope. Variables and functions share the same name space.

Functions

The first form is a procedure declaration (no return type); the second is a function (with return type). Functions return a value of the specified type; procedures are only called for their side-effects. Both forms allow the specification of a list of zero or more typed arguments, which are passed by value.

Standard Library

function print(s : string)
    Print the string on the standard output.
function printi(i : int)
    Print the integer on the standard output.
function flush()
    Flush the standard output buffer.
function getchar() : string
    Read and return a character from standard input; return an empty string at end-of-file.
function ord(s : string) : int
    Return the ASCII value of the first character of s, or −1 if s is empty.
function chr(i : int) : string
    Return a single-character string for ASCII value i. Terminate program if i is out of range.
function size(s : string) : int
    Return the number of characters in s.
function substring(s:string,f:int,n:int):string
    Return the substring of s starting at the character f (first character is numbered zero) and going for n characters.
function concat (s1:string, s2:string):string
    Return a new string consisting of s1 followed by s2.
function not(i : int) : int
    Return 1 if i is zero, 0 otherwise.
function exit(i : int)
    Terminate execution of the program with code i.

Sample Tiger Programs (Scalar Dot Product)

Example I:

```
let
    type ArrayInt = array [100] of int;  /*Declare ArrayInt as a new type */
    var  X, Y : ArrayInt = 10;  /*Declare vars X and Y as arrays with initialization */
    var i, sum : int = 0;
in
     for i := 1 to 100 do                              /* for loop for dot product */
          sum := sum + X[i] * Y[i];
     enddo
     printi(sum);   /* library call to printi to print the dot product */
end
```