Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Assignment - 2: `Lexer for` $\mathcal{F}_{-15}$ *Marks: 100*
Assign Date: *September 5, 2024* Submit Date: *23:55, September 15, 2024*

In a series of four assignments, we intend to implement a compiler for a Fortran-like language. To keep the problem tractable, we present a very small subset $\mathcal{F}_{-15}$ of Fortran 90 that is easy to manage and yet has most of the key flavors of Fortran 90. It is based on Fortran 90 as standardized in Fortran 90, **ISO/IEC 1539:1991**. We present an overview of this language in Section 1. This gives most notions of its syntax and semantics. The details of its lexical and syntactic specifications, framed by stripping down the Fortran 90 standard, are presented in Section 2 after a couple of example programs in $\mathcal{F}_{-15}$ as given in Section 2.1.

The implementation of the compiler for $\mathcal{F}_{-15}$ is split into 4 assignments as follows:

1. **Assignment 2**: Lexical Analyzer for $\mathcal{F}_{-15}$ using Flex. The lexical grammar specification is given here.
2. **Assignment 3**: Parser for $\mathcal{F}_{-15}$ using Bison. The phase structure grammar specification is given in Assignment 2.
3. **Assignment 4**: Machine-Independent Code Generator for $\mathcal{F}_{-15}$ using syntax-directed translation with Bison. Three-Address (intermediate) Code (TAC) used as target of translation is explained here.
4. **Assignment 5**: Target Code Generator for $\mathcal{F}_{-15}$ by simple code generation by table lookup. The target processor is taken to be x86 and a subset of its assembly language is presented here for use.

1 Overview of femtoFORTRAN or $\mathcal{F}_{-15}$

$\mathcal{F}_{-15}$: femtoFORTRAN or $\mathcal{F}_{-15}$, is a femto (10^{-15})-sized language designed based on Fortran 90. It supports the following:

- The *entire program* has a name (`<Program Name>`) and is sandwiched between `PROGRAM <Program Name>` and `END PROGRAM <Program Name>` specifiers. The `<Program Name>` must be identical between the two specifiers.
- *Single line comments* that start with an exclamation (`!`) and end at the end of the line.
- *Variables* and *literals* (*constants*) of 32-bit `INTEGER` *data type*.
- *Variables must be declared* before use. All declarations must be put between the `PROGRAM` line and the first executable statement.
- *Literals* (*constants*) of `string` *data type*. *No variable* of this type is allowed.
- There is *no function*.
- *Lexical Scoping* is supported, but there is *no block scope*. Hence, there is only one *Scope – the global scope* where all variables and compiler generated temporaries belong.
- *Binary arithmetic operations* of addition (`+`) and subtraction (`-`) for `INTEGER` variables and constants. The outcome of an arithmetic expression is of `INTEGER` type.
- *Assignment statement* to assign an arithmetic expression to an `INTEGER` variable.
- *Binary relational operations* between `INTEGER` variables and constants.
  - The outcome of a relational expression is Boolean which can be checked for deciding control flow but cannot be stored in a variable.
  - The relational operators supported are: Greater than (`.GT.`), Less than (`.LT.`), and equal-to (`.EQ.`)
- *Conditional statement* `IF (<Boolean Condition>)-THEN-ELSE` based on a Boolean condition, which is a Boolean expression. The statement ends with `END IF`.
  - A list of one or more statements are placed between `THEN` and `ELSE`. These will be executed if the Boolean condition evaluates to true.

  - A list of one or more statements are placed between `ELSE` and `END IF`. These will be executed if the Boolean condition evaluates to false.

- *Determinate* `DO` *loop statement* with an `INTEGER` *index variable* that runs from an `INTEGER` *lower bound* to an `INTEGER` *upper bound*.
  - The upper bound cannot be smaller than the lower bound.
  - The index variable is initialized with the lower bound at the start of the loop.
  - The index variable is incremented by unity every time the control goes over the loop.
  - Iterations continue as long as the index remains smaller than the upper bound.
  - The loop has a body comprising one ore more statements between the `DO` loop header and `END DO`. These statements are executed every time the control iterates over the loop.
- *Input statement* (`READ`) to read `INTEGER` constants into `INTEGER` variables in free format.
- *Output statement* (`PRINT`) to print `INTEGER` variables and constants, or `string` constants in free format.
- The text of a $\mathcal{F}_{-15}$ program cannot be written in a free formatted manner (like we do in C). Every declaration, assignment statement, and read or print statement needs to be written in separate lines. Conditional and loop statements must be written over multiple lines as can be seen in the samples below.

2 Specification of femtoFORTRAN or $\mathcal{F}_{-15}$

2.1 Sample Programs in $\mathcal{F}_{-15}$

- *Program 1*: To read two integer values and print their absolute difference

```
! Program 1: To read two values and print their absolute difference

PROGRAM absdiff

INTEGER a
INTEGER b
INTEGER c

READ *, a, b ! reads a and b in free format

IF (a .GT. b) THEN
    c = a - b
ELSE
    IF (a .LT. b) THEN
        c = b - a
    ELSE
        c = 0
    END IF
END IF

PRINT *, "Absolute Difference of", a, "and", b, "is", c

END PROGRAM absdiff
```

- *Program 2*: To read a natural number `n` and to print the sum of numbers from `1` to `n`

```
! Program 2: To read a natural number n and
! print the sum of natural numbers from 1 to n

PROGRAM sum

INTEGER n
INTEGER i
INTEGER s

READ *, n

s = 0
DO i = 1, n
    s = s + i
END DO

PRINT *, "Sum of first n natural numbers is", s

END PROGRAM sum
```

2.2 Lexical Specification of $\mathcal{F}_{-15}$

- *Keywords*:
  - The keywords of $\mathcal{F}_{-15}$ are:
    `PROGRAM INTEGER READ PRINT IF THEN ELSE GT LT EQ DO END`
  - The keywords are reserved
  - The keywords are case-sensitive
- **id**: *Identifier or name of a symbol*. An identifier comprises one or more lower-case letters.
- **num**: *Integer constant*. An integer starts with a non-zero digit and continues with any digit. It may have an optional sign (`+` or `-`).
- **str**: *String constant*. A string constant is delimited within a pair of double quotes (`"`) and contains lower-case or upper-case letters, digits or space.
- *Operators*
  - *Arithmetic operators*: `'+'` (addition) and `'-'` (subtractions). Both operators are left associative and have the highest precedence.
  - *Relational operators*: `".GT."` (greater than), `".LT."` (less than) and `".EQ."` (equal to). The result of a relational operation is Boolean (true or false). Relational operators are non-associative. They all have the same precedence which is less than precedence of the arithmetic operators.
- *Delimiters*:
  - `'('` and `')'` (*parentheses*): Used as per syntax (check the grammar rules and the sample programs) or to override precedence / associativity in an expression.
  - `','` (*comma*): Used to separate items in a list.
  - `'*'` (*asterisk*): Used to mark free format for input or output.
  - `'='` (*equal*): Used between LHS and RHS of an assignment.

2.3 Syntax Specification of $\mathcal{F}_{-15}$

- Terminals have already been detailed in the lexical specification.
- The grammar of $\mathcal{F}_{-15}$ is given below:

Sr. #	Productions
01	P → PROGRAM id '\n' LD LS END PROGRAM id
02	LD → LD D
03	LD → D
04	D → INTEGER id '\n'
05	LS → LS S
06	LS → S
07	S → id '=' E '\n'
08	S → IF '(' B ')' THEN '\n' LS ELSE '\n' LS END IF '\n'
09	S → DO id '=' E ',' E '\n' LS END DO '\n'
10	S → READ '*' LI '\n'
11	S → PRINT '*' LI '\n'
12	E → E '+' E
13	E → E '-' E
14	E → '(' E ')'
15	E → id
16	E → num
17	B → E ".GT." E
18	B → E ".LT." E
19	B → E ".EQ." E
20	LI → LI ',' I
21	LI → ',' I
22	I → E
23	I → str

2.4 Support for translation of $\mathcal{F}_{-15}$ programs to TAC (3-address codes)

The semantic specification of $\mathcal{F}_{-15}$ is available in its overview above.

To translate an $\mathcal{F}_{-15}$ program to TAC, we need to extend the TAC, use a few basic data structures, and a few utility functions. We present them here for your action designs later.

- *Extended TAC*: We extend the TAC presented in Module 07 with the following:
  - To input a variable, use `read n`
  - To output a variable, use `print n`
  - To mark the end of the program (and control flow), use `stop`
- *Basic Data Structures*: We need several data structures including:
  - Representation of symbols (`struct symbol`)
  - Representation of symbol tables as lists of symbols (`struct symbolTable`)
  - Global symbol table pointer (`globalST`)
  - Representation of quads (`struct quad`)
  - Representation of array of quads (`quadArray`)
  - Running index on array of quads (`quadIndex`)
  - Representation of labels or indices in quad array (`struct label`)
  - Representation of lists of labels (`struct labelList`)

  You may add more data structures as you need. Briefly explain the purpose of the data structure with its fields.

```
// A symbol record. Note that all symbols are of INTEGER type
typedef struct symbol {
    char *name;          // lexical name of the symbol
    unsigned int width;  // width of the symbol in bytes
    unsigned int offset; // offset in the symbol table
    struct symbol *next; // address on next symbol in the symbol table
} symbol;

// A symbol table as a list of symbols
// F-15 has only one symbol table - the global one
typedef struct symbolTable {
    symbol *first;     // pointer to the first symbol in the symbol table
    symbol *last;      // pointer to the last symbol in the symbol table
    unsigned int nSym; // number of symbols in the symbol table
} symbolTable;

// Global symbol table
symbolTable* globalST;

// A quad
typedef struct quad {
    unsigned int opCode, // encoding of TAC operation - may be binary or
                         // unary or copy or read or print etc.
    const char *result,  // result of the operation
    const char *op1,     // 1st operand - may be null
    const char *op2      // 2nd operand - may be null
} quad;

// Maximum number of quads
#define MAX_QUADS 100

// Array of quads
quad *quadArray[MAX_QUADS];

// Running index on quadArray
unsigned int quadIndex = 100;

// A label record. This is an index in the quad array
typedef struct label {
    unsigned int labelNo;   // label or quad index
    struct list_node *next; // address of the next label record
} label;

// List of labels or quad indices
typedef struct labelList {
    label *head; // first label on the list
    label *tail; // last label on the list
} labelList;
```

- *Utility Functions*: We have the following utility functions on data structures:
  - Create a symbol table (`createSymbolTable`)
  - Look up for a symbol in a symbol table (`lookupSymbol`)
  - Generate compiler temporary (`genTemp`)
  - Add a quad to the array of quads (`addQuad`)
  - Make a list from a label (`makeListLabel`)
  - Merge two lists of labels (`mergeListLabels`)
  - Backpatch a list of labels with a label (`backPatch`)

You may add more utility functions as you need. Specify the prototype of every function you add with brief explanation for its functionality, parameters, and output. No implementation of the function is needed.

```
// Creates an empty symbol table and returns its pointer
symbolTable *createSymbolTable();

// Looks up the global symbol table given a symbol by name
// If the name is found, the pointer to the symbol entry in
// symbol table is returned
// If the name is not found, a symbol record is created, enterted in
// the symbol table, and its pointer is returned
symbol *lookupSymbol(const char *name);

// Generates a temporary variable symbol of INTEGER type
// and looks-up in the global symbol table to enter it
// Generated symbols are serially numbered starting from t00
symbol *genTemp();

// Adds a new quad to the quad array
void addQuad(quad *q);

// Makes a list from a given labelNo
// Returns a pointer to the list
labelList *makeListLabel(unsigned int labelNo);

// Merges two lists of labels
// Returns a pointer to the merged list of labels
labelList *mergeListLabels(labelList *l1, labelList*l2);

// Backpatches a given labelNo to the dangling targets in the list of labels
void backPatch(labelList *l, unsigned int labelNo);
```

3 The Assignment

1. Write a flex specification for the language of $\mathcal{F}_{-15}$ using the lexical grammar. Name of your file should be `FirstName_LastName_A2.l`. *This should not contain the function* `main()`.
2. Write your `main()` (in a separate file `FirstName_LastName_A2.c`) to test your lexer.
3. Prepare a Makefile to compile the specifications and generate the lexer.
4. Prepare a test input file `FirstName_LastName_A2.nc` that will test all the lexical rules that you have coded.
5. Prepare a `FirstName_LastName_A2.pdf` file explaining the working of your lexer and your design choices.
6. Prepare a compressed-archive with the name `FirstName_LastName_A2.x` , where `x` is one of `zip`, `tar` or `rar`, containing all the above files and upload it to Classroom.

4 Credits

1. Flex Specifications: **60**
2. Main function and Makefile: **15 + 5 = 20**
3. Test file: **20**