System Software

Introduction

A computer system performs various tasks as as the hardware executes some software (programs). In the domain of software there are broadly two parts to the solution of a given real world problem. One part concerns about the requirements of the real world problem at hand and the other is about making the solution of the former feasible in the computer which is a complex system in itself. The issues involved in these two parts are significantly different. For a person who creates programs for his own real world problems, it is not feasible to take care of the details of correct hardware operations and optimum utilisation of various system resources (such as memory, CPU time, peripherals, etc.). On the other hand, a person who understands the underlying features of a computer system can take care of this on behalf of different software developers using some general software modules. Hence software for these two parts are handled individually. We find software such as MS Office package, Netscape browser, Oracle DBMS, etc., that take care of some real world problems. These are called

application programs

. On the other hand we find software such as UNIX OS, Windows OS, C Compiler, bash shell, etc. that help users run the application programs in their computers. These software are called

system programs or system software

.

> System software is that software which helps an average computer user's program to execute effectively on a computer system. They address issues which exist due to the computer system, and thereby make the computer "usable" for various real world problem solving tasks.

Lexical Analyzer

Introduction

In the process of translation (of programs) lexical analyser scans an input text and recognises sequences of bytes in it as certain valid "items" or invalid sequences. For example if an assembly language allows a set of mnemonics, alphanumeric identifiers, numbers (sequence of digits), and some symbols such as comma, colon, etc., then a lexical analyser for that assembly language should detect such items in an input program written in that assembly language. Usually a larger assembler program invokes a lexical analyser module to recognise such items one-by-one, starting from the beginning of the text. Each such distinct item that the lexical analyser may detect and pass on (

return

) to the main assembler is called a

token

. So for an assembly language there can be tokens such as IDENTIFIER, NUMBER, LITERAL, COMMA, KEY_LDA, KEY_START, KEY_LOOP, (the prefix KEY_ may indicate that the token represents a keyword) and so on. The designer of a translator can define the set of tokens for his purpose. In an assembler, distinct numbers are used to denote each token, so that the lexical analyser can pass the number representing the token that it recognises in the input. For example, for the above set of tokens we may use 1 for IDENTIFIER, 2 for NUMBER, 3 for LITERAL, 4 for COMMA, 5 for KEY_LDA, and so on. From implementation point of view, in C language one can use the C-preprocessor's define statement to establish these equivalence, such as, #define IDENTIFIER 1

Implementation

The sequences of bytes that a lexical analyser has to recognise are either fixed or variable (

generic

). Fixed sequences are for keywords and some symbols such as comma, colon, etc. where a predefined sequence of one or more bytes (characters) are recognised as some item. For example, for each keyword the sequences of letters are predefined. Variable or generic sequences are for identifiers, numbers etc. In these some rule is given instead of any predefined sequence. An example of a rule is, "a sequence of alphanumeric letters of an underscore, but first letter is not numeric is an identifier". Usually, if a sequence of input bytes matches a fixed sequence as well as some rule, that sequence is

recognised as the item corresponding to the fixed sequence. For example the sequence "MULT" may match the fixed sequence for a keyword as well as the rule for identifier given above. This will be then recognised as a keyword. (Relate this to the concepts of

Keyword

and

Reserved word

.) Some items for recognition may have combination of fixed part and variable part. For example, suppose in a language an address value is prefixed by an "@" character to distinguish it from a numeric constant. Then the rule for an address value is "@ followed by a sequence of digits".

First Approach - String comparision

One approach that can be easily imagined is to read some number of characters form the input file and match the string against some known strings (spellings) of items. If the input string does not match any of these known fixed strings, then it may be further tested to match variable sequences. In practice, most programming languages allow identifiers (names of variables, constants, procedures, etc.) to consist of sequences of alphanumeric and possibly "underscore" characters. The keywords are usually strings of alphabetic characters. Numbers are usually strings of digits (decimal or hexadecimal) and possibly a period (decimal point). Other valid items are formed with other symbols than these mentioned characters. Fixed strings corresponding to keywords can be stored in a table, preferably arranged in such a way that facilitates easy search (eg. sorted to facilitate binary search). Thus, wherever in the input there is a continuous string of alphanumeric characters or underscore, a lexical analyser can read it into a buffer. While doing so set a boolean flag to indicate whether any underscore was read, another to indicate whether any alphabetic character was read, and another to indicate whether any decimal digit was read. If in the string there was no underscore or digits, then the string might be a keyword. The string is matched against the keywords stored in the table. If a match occurs, the lexical analyser declares the string to be that keyword. Otherwise it is an identifier. If the string in the buffer contained one or more underscores, it is not necessary to search the keyword table, and strightaway it can be recognised as an identifier. If however, the string had only digits it can be recognised as a number. In doing so some more characters from the input may also be included if there is a period followed by some more digits.

Other items of the language may be recognised by matching the input character-by-character. The above is a description of a

possible

lexcial analyser for

typical

programming language. Variations of it can be implemented for any real requirement.

Second Approach - Finite State Machine

The first approach described above is easy to conceptualise and can be implemented as a program without much diffuculty. But the need to search words in the table of keywords, and other matching operations in the remaininig part,

essentially

requires each input character to be compared with constants multiple number of times. (

Exercise : Why is each character compared with constants multiple times ?

) This situation may well be improved in terms of computational complexity. The second approach to building a lexical analyser is using a

Finite State Machine (FSM)

. It reduces the computational complexity such that each input character is to be handled only once (in fact, there is no comparision at all). In an FSM a set of

states

are defined for the system. Initially the system assumes a pre-defined state called the starting state and as the input is processed, the system makes transition to another state (transition to the same state is also possible) for each input symbol. In a way, a state actually indicates what sequence of input symbols have been seen so far and what

action

to take as a result. The action may be simply make another state transition using the next input character, or declare that a token has been recognised in the sequence of input symbols. In the latter case the system again assumes the starting state so that it can proceed to process the remaining input characters to recognise more tokens. Usually, the action associated with each state also

includes some other tasks, such as to update some program data structures, etc.

For different languages the FSM for recognizing the tokens will vary in the set of states, the state transitions that are to be made, and the tokens to be recognized. It needs to be understood that the implementation of an FSM as a program has two parts - some *static* data that describes the states and the state transitions, and a set of instructions for the program to proceed through a series of state transitions. The data is structured as a *state transition table* where each row stands for a state of the FSM and each column stands for an input symbol. Each entry in the table indicates the *state that the FSM should enter from the current state (corresponding to the row) upon encountering the input symbol corresponding to the column*. With this information in the table, the code of FSM should simply facilitate the selection of the next state corresponding to the current state and the next input symbol by a look up of the table. However, other actions associated with each state also needs to be incorporated.

The tedious part in implementing the FSM model as a lexical analyser program for a particular language is to determine the states and the state transitions. Fortunately there exists tools that makes this very simple. One such tool is *lex* in UNIX (*flex* in Linux). This tool takes the lexical specifications in a file, in the form of regular expressions corresponding to each token to be recognised. The actions to be taken upon recognition of each token should be specified along with the regular expression for that token. The tool then creates a C program file that essentially contains a lexical analyser function named *yylex()*, which can be called to perform lexical analysis according to the specifications encoded. Thus *lex (or flex) is a tool that creates a lexical analyser using lexical specifications and actions corresponding to tokens provided as input.* [See *man flex* for more details]

Assembler Data Structure

```
A DESIGN OF ASSEMBLER DATA STRUCTURES
[The following (till the line "Assignment 1 ends") is a sui
table response to
the Assignment 1 given during Sept 2000.]

/* The following data structures may be used in a 2-pass as
sembler - */

/* Mnemonic Table : */
#define MAX_MNE_LEN 6   /* max no of char in a mnemonic opc
ode */
#define MAX_OPRND   2   /* max no of operands in an instr
*/
typedef enum mne_class {
        IS,         /* Imperative statement */
        DS,         /* Storage declaration statement */
        AD          /* Assembler directive statement */
    } mne_class_t;
typedef enum oprnd_typ {    /* Operand types - depends on t
he language */
        NONE=0,
        REG,        /* Register operand */
        NUM,        /* Numeric operand */
        ADDR        /* Address operand */
    } oprnd_typ_t;
typedef struct opcod {
        char    mne[MAX_MNE_LEN+1];
        mne_class_t type;
        oprnd_typ_t oprnd[MAX_OPRND];
        short   len;        /* no of bytes */
        int     (*fn)();    /* function corr to Asslr.Dir.
*/
    } opcod_t;
extern opcod_t  optab[];    /* To be initialised during def
inition */
extern const int max_opcodes;   /* To be initialised during
definition as - */
                /* (sizeof( optab ) / sizeof( opcod_t )) */
```

```
#define MAX_OPCODES (max_opcodes)


/* Symbol Table */
#define MAX_SYM_L   15  /* max len of a symbol */
typedef long    addr_t;
typedef enum {
        DATA,       /* Symbol represents a data addr */
        CODE,       /* Symbol represents a code addr */
        OTHER
    } obj_typ_t;
typedef struct symbol {
        char    str[MAX_SYM_L+1];
        addr_t  address;
        obj_typ_t   type;
        short   size;   /* valid only for type==DATA */
    } symbol_t;
#define MAX_SYMBOLS 50
extern symbol_t symtab[MAX_SYMBOLS];    /* Symbol table !
*/
extern short    n_sym;      /* no of syms in symtab : idx f
or next */

/* Literals Table */
typedef enum {
        NUM_1B, /* Single byte number */
        NUM_2B, /* 2-byte number */
        NUM_4B, /* 4-byte number */
        STRING  /* String */
    } lit_type_t;
#define MAX_LIT_L   20  /* a literal string can be upto 20
bytes */
typedef struct literal {
        lit_type_t  type;
        char    value[MAX_LIT_L+1];
        addr_t  address;
    } literal_t;
#define MAX_LITERALS    50
```

```
#define MAX_LIT_POOLS   20
extern literal_t    littab[MAX_LITERALS];   /* Literals tab
le ! */
extern int      pooltab[MAX_LIT_POOLS]; /* Literal pools ta
ble ! */
extern short        n_lit,  /* no of literals : index for n
ext */
            n_pool; /* no of lit pools : index for next */

/* Intermediate code record */
typedef struct oprnd {
        oprnd_typ_t type;
        int     value;  /* Reg #, symtab/littab/pooltab id
x, value */
        short   size;
    } oprnd_t;
typedef struct icr {
        short   opcod_idx;  /* index to optab */
        oprnd_t oprnds[MAX_OPRND];
    } icr_t;
/*
 * Intermediate code records may be prepared for each instr
uction in a buffer
 * and then written to an intermediate-code-file.
 */

/* Location Counter */
extern short    loc_cntr;

Usage of the Above Data Structures -
====================================

1. Mnemonics Table optab[]  :

    Initialisation -  The  optab  is  initialised  statical
ly,  i.e.,  during
    definition. It is filled  with  the  opcode  mnemonics
of  the  assembly
```

language. It is not changed when the assembler works.

Reference - During pass-I of the assembler, optab[] is accessed to determine the attributes of the mnemonic opcode found in an assembly statement. If the lexical analyser is constructed using a tool such as lex, then the lexical analyser can be made to directly tell the position of an opcode in the table. Otherwise, the position may be determined by performing a search on the mne field of the records. In that case, it is useful to have the entries of the optab sorted before hand. In pass-II optab is accessed to determine the machine opcodes corr esponding to the imperative statements in the intermediate code.

2. Symbol Table symtab[] :

Pass I - The symbol table is built up during pass-I of the assembler. The number of entries in symtab at any time is indicated by the value in the variable n_sym which is initially set to 0 and increme nted upon making each new entry in symtab. Whenever a symbol is en countered in the program text, it is searched in the symtab. If the curr ent occurrence of the symbol is its definition and it is not already ther e in the symbol table, then a new entry is made in the symtab specifyin g all the fields, including the location counter value as the address of the symbol. If the

```
    symbol denotes data the size if known from the statemen
t. If  the  symbol
    denotes code address, the size field may be left  unini
tialised.  If  the
    current occurrence of the symbol is its definition and
the symbol already
    has an entry in the symtab, then  the  address  field
of  the  entry  is
    updated using the current value of the location counte
r.  Also  the  size
    field may be updated according to  the  current  statem
ent  defining  the
    symbol. On the other hand, if the current occurrence  o
f  the  symbol  is
    only a reference, and it not already present in symtab,
a  new  entry  is
    made in symtab without specifying the address field  (s
ince  the  address
    will be known only when it is defined later). If the  c
urrent  occurrence
    of the symbol is only a reference, and it is already  p
resent  in  symtab
    the entry need not be updated. In both these cases of
symbol  reference,
    the index of the symbol's entry in the symtab is used i
n the Intermediate
    code that is being produced.

    Pass II - In intermediate code produced in  pass-I  con
tains  indices  of
    various entries of the symtab. In pass-II,  while  prod
ucing  the  target
    code the symtab references in the intermediate code are
replaced  by  the
    address field contents  of  the  corresponding  entries
in  the  symtab.

3. Literals Table littab[] and pooltab[] :
```

Pass I - The literals table littab[] is built up during pass-I of the assembler. The number of entries in littab at any time is indicated by the value in the variable n_lit which is initially set to 0 and incremented upon making each new entry in littab. Whenever a literal is encountered in the program text, it is entered in the littab and the position (index) of the entry in the littab is recorded for the operand in the intermediate code. The number of literal pools is indicated by the value in the variable n_pool which is initially set to 0 (one less than the number of pools). Upon encountering an LTORG or the END statement addresses are assigned to each literal in the littab that are not yet assigned addresses, i.e., the current pool. The value of the location counter (loc_cntr) is used as the address of each such literal and this value is incremented according to the size (number of bytes to be occupied by each literal) of each. The variable n_pool is incremented to indicate that one pool is over. When the value n_pool is set to 0 and subsequently whenever it is incremented) to indicate the end of a pool, the entry pooltab[n_pool] is set to the current value of n_lit. Thus the entries of pooltab contains the starting positions of each literal pool.

Pass II - In intermediate code produced in pass-I contains indices of various entries of the littab. In pass-II, while producing the target code the littab references in the intermediate code are replaced by the address field contents of the corresponding entries in the littab.

4. Intermediate Code :

Pass I - Intermediate code is produced during pass-I. In the intermediate code, for each source statement, the mnemonic opcode is replaced by the opcode's entry number in the optab[]. For imperative statements, the symbols or literals used as operands are replaced by their corresponding entry numbers in the symtab[] and littab[] respectively. No label field is present corresponding to the source text. The intermediate code record are of fixed sizes and contains room for details of MAX_OPRND (two) number of operands. If a statement has fewer operands, the "type" field of the unused operands is set as NONE. Intermediate code record prepared for each source statement is written to an intermediate code file.

Pass II - Intermediate code records from the file are read one at a time in pass-II and target code is produced. The variable loc_cntr is initialised to 0 in this pass and incremented according to the size of

```
    each target statement produced. This variable is also
updated  according
    to the operand of  START  and  ORIGIN  statements.  A
variable  pool_idx
    (literal pool table index) is initialised to 0 and incr
emented  for  each
    LTORG statement encountered  in  the  intermediate  cod
e.  The  value  of
    pool_idx denotes the current  literal  pool  whose  sta
rting  and  ending
    littab indices are pooltab[pool_idx] and (pooltab[pool_
idx+1]-1). Against
    an  LTORG  or  END  statement  in  the  intermediate
code  the  actual
    representation of the literals in the current pool  is
produced  as  the
    corresponding target code. Apart from the literals, the
re  is  no  direct
    representation of the Assembler directive statements in
the target  code.
    The target code produced against  each  intermediate  c
ode  statement  is
    written to a target code file.
```