Compiler Project 2 Report

Yinsong Xue

5120309415

Part I Features

I believe I have implemented nearly every requirement except code optimizing.

The core components of my project are:

1. A parser and intermediate code generator which is generated by lex and yacc. In this part I move some parts of the code to other c source files(sym_table.c/h etc.)
2. A simple intermediate scanner which performs some very simple operation to prepare for the next stage.
3. A target machine code generator which transforms the Mips-like intermediate code to actually Mips code and performs the register assignment actions.

The first part is written in lex and yacc format.

The second and third part of my project is written in Python.

The bonus features include but not limited to:

- Deal with octal and hexadecimal numbers.
- Type-check(which I believe is complete)
- Check types and numbers of function arguments.
- Check the numbers and types when initiating an array.

Part II Parsing and Intermediate code generating

To parse a SMALLC source file, we need to change some of the evaluation rules to solve certain difficult situations.

1. In order to use inherited attributes, we need to rewrite most of the evaluation rules. For example, the rule
   STMT -> FOR LP EXP1 SEMI EXP2 SEMI EXP3 RP STMT
   Should be changed to

FOR LP MF2 EXP SEMI M26 EXP SEMI M26 EXP M66 RP M26 STMT.

2. In order to deal with arrays, we should rewrite the array initial statements.
3. If we want to perform an assignment, we have to define a Leftvalue symbol.

The data structures I used in my code are:
Symbol_table --- which is stores symbol_table information.
Type --- which stores all information about types.
Queue --- which stores function arguments.
All information of node in an annotated parse tree is stored in the following structure.

```
typedef struct {
    int Intval;
    char * Strval;
    TYPE *Type;
    TYPE *TarType;/* used for arrays*/
    TYPE *array;
    int array_place;
    int ndim;
    int offset;
    int *true_list;
    int *false_list;
    int *next_list;
    int *break_list;
    int *continue_list;
    symbol_table *s_table;
    Place_type quad;
    int place;
    int num;
    queuetype *queue;
    int sdecs_dest;/*0-derived from struct declaration
                     1-derived from struct varible declaration
                        */
}NODEtype;
```

PART III. Intermediate code

When the parsing process finished, the intermediate code will be stored in InterCode.txt. The format is similar to MIPS, but with the following information.

1. All the jump destinations are represented in raw line numbers, which is very unstable.
2. A total line number and a total used-variable number is carried in the first line of the Intermediate code, because they would be needed in the upcoming processes.
3. All the arrays, global variables and some of the local variables will have a

declaration statement of format'def(arr) varname varsize' .

Though this is an acceptable intermediate code format, but the jump destinations and variable definitions are annoying, so I use splitcode.py to scan the whole intermediate code and do the following things and save the result to Intercode1.txt.

1. Recognize whether a declared variable is local or global. If it's local we add the function that it belongs to to the declaration.
2. For each jump operation, it find out the jump destination and rename them to L+linenum, then add 'Label L_linenums' to the code.

Part IV. Machine-code Generation

In this part, the generator has a lot of things to do. Generally the program scans the intermediate code function-by-function, and it will scan only once.

1. Get the global variables' information and store them.
2. Rename all the variables because they all are of the routine 't0,t1…', if they are global variables , change 't' to 'G', if they are temporary variables, change them to 'T'.
3. In each function:
    a) For every global variable or stack variable(variables which stores in stacks) each time we approach them, we should first load them to a temp variable, then save the temp variable back to stack / data area.
    b) If the code is of format 'sw/lw t0 t1(t2)', we should change it to following codes.
    Mul t1 t1 4 #every int takes 4 bytes
    La t3 t2 #notice that t2 must be a global/stack variable
    Add t1 t1 t3 # calculate the address with offset
    c) Every other pseudo-codes(return / read/ write etc.) are translated to MIPS codes.
    d) After c, we store the new intermediate code to intercode2.txt (not necessarily, just for checking)
    e) Then we do the register allocation, first we should scan the function and divide it to basic blocks, notice that since function calls my change t0-t9, so every instruction after a function call should also be set to a leader. Then we calculate the jump relations between blocks.
    f) We scan the function to get all the variables that need to assign, meanwhile we collect the information of the variable in each block(the first line it is needed, whether the variable needs to exist in the first line of the block…etc)
    g) Combine the results of e and f together we can get the alive range for every variable.
    h) Use linear scan algorithm to allocate the variables.
    i) If a variable has to be in stack, jump to e
4. After step 2, we nearly get the target code right. We just have to do some

final actions. Like change the names of the global variables, save/load the stack variables.

5. A small note: Since the main function needs to use syscall to exit but currently uses 'jr $ra', I rename the main function to "ACTUAL_MAIN", and creates a new function main which calls ACTUAL_MAIN.

Part V. Summary

This is one of the most challenging project I have ever solved. Maybe I failed to submit it in time, but I myself has learned a lot in this project. I'm happy I got the chance to implement a compiler myself. Thanks to Prof. Wu and the T.As.

Yinsong Xue