# CS215: Implementing a Compiler

朱佳顺 **5100109194**

## contents

1 Introduction ..... 2

2 Intermediate code ..... 2

3 Expression Calculation ..... 3

4 Global variables and Symbol Table ..... 5

4.1 Global variables ..... 5

4.2 How to make up the symbol table ..... 6

5 Function ..... 7

5.1 Function Call ..... 7

5.2 Activation Record ..... 8

6 Type Checking and Error Checking ..... 9

6.1 checking redefined element ..... 9

6.2 checking assigning exp to left-value ..... 9

6.3 checking variable/function without declaring first ..... 9

6.4 checking wrong array initialization ..... 9

6.5 array checking ..... 10

6.6 struct checking ..... 10

7 Test ..... 11

8 pains and gains ..... 11

9 Acknowledgement ..... 12

# 1 Introduction

During the previous project, we have been asked to complete the Syntax analysis and Semantic analysis. In this project, we are asked to produce the intermediate code, optimizing the code and finally generating the real Mips code to run on the spim simulator. Perhaps it is the most difficult part to generate the Mips code. I have used several weeks to think about the appropriate data structures to fit my thought. In the following part, I will introduce how I implement my compiler step by step. It is a really complex job. I wrote more than 2000 lines code for the project.

# 2 Intermediate code

We are recommended two kinds of intermediate code, one is AST, and the other is three-address code. In my opinion, I think three-address code is not so important as syntax tree, So I select syntax tree as my intermediate code.

```
8   class treeNode
9   {
10  public:
11      string data;
12      int childnum;
13      int type;
14      treeNode** child;
15      treeNode(string str, int n, int t=0)
16      {
25      void Clear(treeNode *t)
26      {
34  };
```

I define a class named **treeNode** to represent each node of the tree.

**Data**: the actual data of the node;

**Childnum**: the number of child of the current node.

**Type**: indicate which reduction is taken place.

**Child**: is the array storing the pointer to its child

Using yacc I can get the root of the entire tree from the source program. When I have the root, then I can pretravel the tree, I also can generating the Mips code depending on the parse tree.

# 3 Expression Calculation

How to calculate the Exp bothered me a lot. Because In class, prof. Fwu taught not a lot about how to effectively select registers. Since there are a lot of Register that can be used, I think for Small C , two register is totally enough. As a result, I only use $t0 and $t1 as my calculation registers as well as $sp, $s0, $s2 (later be explained.) One important thing that I must do is to put the previous $t0 into the stack so that I can use $t0 again to do some calculation, eg: 2*3+4*5: here is what my program will do: put 2 into $t0, 3 into $t1, mult them into $t0, store $t0 into stack, then put 4 into $t0, put 5 into $t1, mult them into $t0, load the top element in the stack into $t1(which is 2*3 = 6), then add them put the resule into $t0.

**Remenber: I always keep the current value in the register $t0, which simplified the problem in a great way.**

Here are some code examples:

Case 1 means take the first reduction. The result is in the $t0.

```
734  void genExp(treeNode* node, int type, string scope, int pre)
735  {
736      switch(type)
737      {
738          case 1:
739          genExp(node->child[1], node->child[1]->type, scope);
740          cout << "seq $t0, $t0, 0" << endl;
741          break;
742
743          case 2:
744          genExp(node->child[1], node->child[1]->type, scope, 1);
745          //cout << "add $t0, $t0, 1" << endl;
746          break;
747
748          case 3:
749          genExp(node->child[1], node->child[1]->type, scope, 2);
750      //   cout << "add $t0, $t0, -1" << endl;
751          break;
```

Some detail explanation:

I use $t0, $t1, $t2, $s0, $s2 totally.

**$t0** is always the register which the current Register put.

**$t1**, $t2 are a temporary registers.

**$s0** is the base register for para and locals

**$s2** is the base register for global. Eg: lw $t0, 4($s2), which means put the value in the address $s2 + 4 into $t0.

# 4 Global variables and Symbol Table

## 4.1 Global variables

Here is the global variable I use in this project:

```
extern string outputFile;
extern int labelNumber;
extern int globalNumber;
extern map <string, map<string, int>*> FT;
extern map <string, map<string, int>*> ST;
extern map <string, string> STAC;
extern map <string, map<string, int>*> ISA;
```

**outputFile** is a string variable to keep record of the outputfile name.

**labelNumber** is a variable to make every label unique and trackable.

**globalNumber** is a variable to label the global variable when initialized.

The following four STL map is the Symbol Table of my program, which is the most important part when generating the Mips code.

**FT**: It is hash table. The first string is to index the function. The second string is used to index the parameter or the local variables in this function, and the int value is the offset used with $0 to find the location of the variable in the stack. Particularly, FT[“_G”] is used to index global variables.

**ST**: similarly as FT. The first string is to index the struct name. The second string is used to index the attribute of the this struct, and the int is the offset used to find the specified attribute.

**STAC**: the map from specified struct to abstract struct.( Eg: struct name id; then

STAC[“name”] = “id”. ) It is also used to type checking.

**ISA** : meas Is Array or not. The first string is to index the function, since we can define array in every function. The second string is the variable name. If (*ISA[“main”])[“a”] = 1, it means that a is an array in the function main. It is also used to type checking.

## 4.2 How to make up the symbol table

Whenever I have the root of the tree, everything becomes easy. I should tranverse the tree in the First Pass, put all the variables and the offset into the symbol table, when I want to retrieve a value, the only thing I will do is to check the symbol table to type checking and get the offset from $0(which is a base register to index variable.)

Here is some code:

This is used to construct the global Symbol Table:

```
11  void genGT(treeNode* node)  //for global construct symboltable
12  {
13      //int a,b,c[10] ÀàÐÍµÄÇé¿ö
14      if(FT["_G"] == NULL)
15      {
18      if(ISA["_G"] == NULL)
19      {
22      if(node->child[0]->type == 1)
23      {
80      //struct opt {DEFS} modify ST
81      else if (node->child[0]->child[0]->type == 1)
82      {
159     //struct id xxx;
160     else if(node->child[0]->child[0]->type == 2)
161     {
204 }
```

This is used to construct the Function Symbol Table:

```
206  void genFT(treeNode* node)  //
207  {
208      int size = 0;
209      if(FT[node->child[1]->child[0]->data] == 0)
210      {
213      if(ISA[node->child[1]->child[0]->data] == 0)
214      {
217
218      else
219      {
226
227
228      string tmp = node->child[1]->child[2]->data;
229      if(tmp[tmp.length()-1] != ',')
230      {
233      if(node->child[1]->child[2]->type != 3)
234      {
253
254      int localn = 0;//¼ÇÂ¼¾Ö²¿±äÁ¿µÄ¸öÊý
255      tmp = node->child[2]->child[1]->data;
256      if(tmp[tmp.length()-1] != ',')
257      {
260      if(node->child[2]->child[1]->type == 1)
261      {
319      (*FT[node->child[1]->child[0]->data])["#"] = size*4;
320      (*FT[node->child[1]->child[0]->data])["##"] = localn;
321
322  }
```

After constructing the Symbol table, I can generate some real code.

# 5 Function

## 5.1 Function Call

This is how I implement function: put all the parameters into the stack. Reserve the places for local variable. Store previous $s0 into $sp-4, then Put $sp-4 into $s0, which is a new value. Jump to the Function Code, after executing the function, jump back and restore $s0 and $sp. ($s0 is used to index variable by the offset in the Symbol Table.)

The logic is quite easy and simple.

Here is some core code:

```
cout << "add $sp, $sp, -" << (*FT[node->child[0]->data])["##"] * 4 << endl;
cout << "add $sp, $sp, -4" << endl;
cout << "sw $s0, 0($sp)" << endl;
cout << "move $s0, $sp" << endl;
cout << "jal " << "__" << node->child[0]->data << endl; //Ìø×ª
cout << "add $sp, $sp, " << 4+(*FT[node->child[0]->data])["#"] << endl;
cout << "lw $s0, 0($s0)" << endl;
```

## 5.2 Activation Record

Here is the activation record when the main Function calls another function. From the picture you can see that previous $s0(0x10) is stored in the new $s0(0x04) position, which is used to restore the $s0 when the function returns. And the Return address is right after the $s0. It will be easy to return from the address. Because the implementation is quite complex, I can't explain all the details in this report.

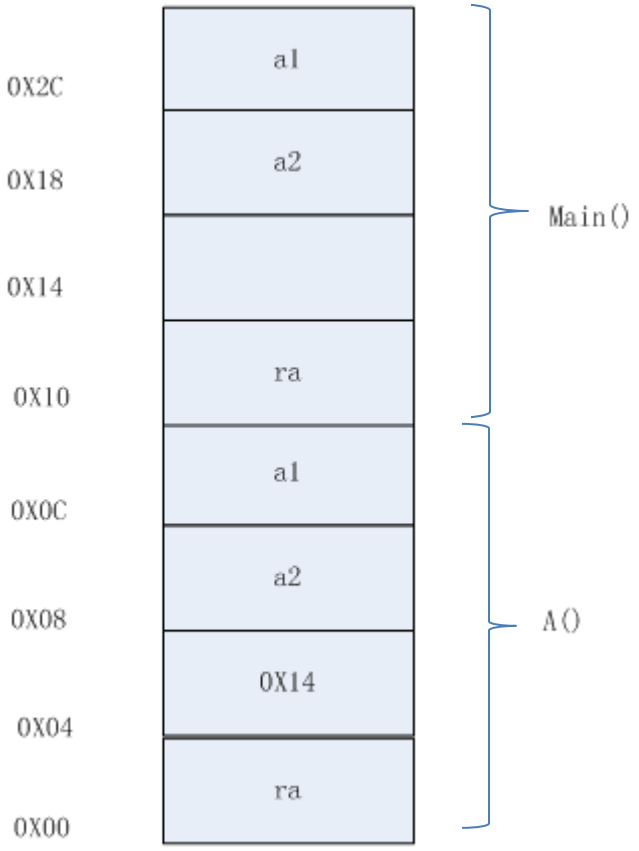


Fig 5.2

# 6 Type Checking and Error Checking

## 6.1 checking redefined element

The errors of redefining in the Fig 6.1 can all be detected at compile time.

```
1  int a;
2  int a[10];
3  or
4  int a; int a;
5  or
6  struct id a;
7  int a;
```

Fig 6.1

## 6.2 checking assigning exp to left-value

Codes like

```
5  int a;
6  1 = a;
```

will be detected at compile time and report an error.

## 6.3 checking variable/function without declaring first

Code in Fig 6.3 assign a value to a variable without declaring is not allowed. Such Error will also be detected.

```
1  a = 1;
2  b[1] = 2;
3  st.aa = 3;
```

Fig 6.3

And similarly, any function without declaring is not allowed and report an error.

## 6.4 checking wrong array initialization

Error in Fig 6.4 such as (1) initialize a value to an array, or (2) initialize too many elements than the size of the array will be detected at compile time and report an error.

```
1  int a[5] = 1;
2  int a[5] = {1,2,3,4,5,6};
```

Fig 6.4

Here is some concrete codes to implement this idea:

```
if(size > n)
{
    freopen(outputFile.c_str(), "w", stdout);
    cout << "Error.";
    fprintf(stderr, "\nCompile fail!\n");
    fprintf(stderr, "error occurs, %s argument is too large\n"
    exit(0);
}
```

## 6.5 array checking

Only array element can use [] to index number. Data structure **ISA** is well-designed for this purpose. If the corresponding int number in ISA is 1, then the array operation is allowed. For example, the error in the Fig 6.5 will be detected at compile time and report an error.

```
1  int a;
2  struct name b;
3  a[1] = 1;
4  b[2] = 3;
```

Fig 6.5

## 6.6 struct checking

Only struct element can use . to index attribute. Data structure **STAC** is well-designed for this purpose. The reason is quite similar with 6.5. Code in Fig 6.6 will be detected and report an error.

```
1  int a;
2  a.sss = 1;
```

Fig 6.6

# 7 Test

I have written code to test all my operators, such as +=, <<=, %, *=, ++ , --…. All the operators appearing in the Gammar are tested and the result is in my expectation.

I also test all the pretests given by our TA and pass them all.

It is a great pleasure to complete my compiler within 2 weeks. And I also spend plenty of time in it.

The whole Project is really huge, I can't put all the details. Maybe I will cover them in my presentation.

# 8 pains and gains

1) It is very annoying to select a good data structure used to act as symbol Table. Finally I use STL, and it is easy to use.

2) How to deal with recursive function call. The Logic is easy. But in practice, I debug for a while to pass the test.

3) How to deal with local variables annoys me a lot. But locals are the callee's responsibility to push them into the stack. But in my Compiler, I give this work all to the caller to simplify the problem in a great way.

4) How to initial the value. This doestn't bother me a lot because I have a good

data structures.

5) How to debug is a really important skills. In the project, I practice more and more using “printf” to debug my program. GDB is not recommended.

6) In this project, I wrote more than 2000 lines code, which improved my ability of programming in a significant way. How to select a good data structure is also a very important thing.

# 9 Acknowledgement

Thanks to Prof.Fwu teaching the class for us and help us have a thorough understanding about compilers.

Thanks to our TA GuoTeng Rao, without whose effort my compiler will never complete. He answered me a lot of questions by QQ. I really appreciate that.