Compiler Project 3

11611006 邓韵杰

In this project, we will generate a intermediate codes for our spl program, which can run on a simulator.

Basic requirements

According to required test cases, we only need to implement logic jump, function, assign operation and arithmetic operations. `struct` and `array` is base on address. Either of them is not used in basic requirements.

read() and write()

These two functions are in our symbol table, although both of them is not defined in our program. Notice that the way to call these functions are different from other functions. Hence, we should carefully deal with them firstly.

Function

We can use following code to define a function

```
FUNCTION funcName :

PARAM p1

PARAM p2

PARAM p3
```

, where `funcName` is the name of function and `p1`, `p2` and `p3` are parameter of this function. Since function is global, we can the original function name to define the function in IR. (For variable, we may need to consider its scope.)

We do not need to implement the detail of function call by ourselves, and we use following instruction to call a function

```
ARG a3
ARG a2
ARG a1
t := CALL funcName
```

, where `funcName` is the name of the function and `a3`, `a2` and `a1` are input arguments when call the function. Notice that the order of arguments is in reverse.

Arithmetic operations and Assign operation

We can translate our program to by using corresponding instruction in proper time.

The implementation of this part is not hard. But there are some weird things when I use instructions. When I use instruction to calculate `2001/100*100`, which can be calculated in following instructions

```
t0 := 2001
t1 := 2001 / 100
t2 := t1 * 100
```

. The result in t2 is still 2001. I think `urwid` instructions treat the number as float number, which quiet confuses me in this project. And I do not know how to deal with this thing.

Logic jump

I think the hardest part in basic requirement is logic jump. We should use back patching to deal with logic jump.

We should add some mark in logic jump statement to do some semantic action.

```
Stmt -> IF LP Exp RP L1 Stmt L2{
            backPatchList(Exp->trueList, L1->inst);
            backPatchList(Exp->falseList, L2->inst);
        }
        | IF LP Exp RP L1 Stmt G ELSE L2 Stmt L3{
            backPatchList(Exp->trueList, L1->inst);
            backPatchList(Exp->falseList, L2->inst);
            backPatch(G->inst, L3->inst);
        }
        | WHILE L1 LP Exp RP L2 Stmt G L3{
            backPatchList(Exp->trueList, L2->inst);
            backPatchList(Exp->falseList, L3->inst);
            backPatch(G->inst, L1->inst);
        }
```

Notice that `L` is an empty rule used to generate `LABEL` instruction. And `G` is also an empty rule, but used to generate `GOTO` instruction.

```
G -> %empty {generate GOTO ____}
L -> %empty {generate LABEL ...}
```

Next, we deal with `Exp` in logic jump. We define `trueList` and `falseList` for `Exp`. `trueList` and `falseList` stores all `GOTO` instructions that do not have any label to go, and the missing label will be back patched in proper time.

```
Exp -> Exp1 AND L Exp2 {
        backPatchList(Exp1->trueList, L->inst);
        Exp->trueList = Exp2->trueList;
        mergeList(Exp->falseList, Exp1->falseList, Exp2->falseList);
    }
    | Exp1 OR L Exp2 {
        backPatchList(Exp1->falseList, L->inst);
        Exp->falseList = Exp2->falseList;
        mergeList(Exp->trueList, Exp1->trueList, Exp2->trueList);
    }
    | Exp1 [relop] Exp2 {
        // next instruction    : IF Exp1 [relop] Exp2 GOTO ____
        // next instruction + 1: GOTO ____
        $$->trueList.add(next instruction);
        $$->falseList.add(next instruction + 1);
    }
```

Function `backPatchList()` will patch all `GOTO` instruction in this list with a label.

With above actions, we can generate instructions for all logic jump.

Bonus implementation

For bonus implementation, we should support `struct` and `array`.

We need to use `DEC name [size]` to allocate space. Before allocate space, we should calculate `struct` or `array` size.

Notice to identify pointer and value of a derived type.

Code Optimization

1. Pre-compute constant `Exp`

   For example, `a = 5 + 5;`. We can directly generate instruction `a := 10` by doing pre-computation during compiling.

2. `GOTO` and `LABEL` optimization

   Sometimes, we can merge some labels together. Notice that we should change corresponding `GOTO` instruction.

   ```
   // before merge
   ...
   LABEL l1 :
   LABEL l2 :
   ...

   // after merge
   ...
   LABEL l1l2:
   ...
   ```

   Or we can delete `GOTO` and `LABEL` together in some cases.

   ```
   // before delete
   ...
   GOTO l1
   LABEL l1 :
   ...

   // after delete
   ...
   ...
   ```