Project 4 Documentation

Thomas Trask

COP4620

Spring 2013

# Project Description

The purpose of project 4 is to consume the tokenized source code created by Project 1 and develop intermediate code in the Quadruple format.

# Project Specifications

The project should do the following things:

* Read a file, line by line, as provided by the first CLI argument for the program.
* Split the file into a series of Tokens, as outlined by Project 1.
* Traverse the series of tokens linearly and generate Intermediate source code on the fashion:
  + <Call Symbol> <param1> <param2> <output>

# Execution

## Command line arguments

The application takes 1 command line argument: the input file that the program should process.

## Executing the application

The application can be run automatically with the following command

./p4 <testfilename>

java Proj4 <testfilename>

# How the program works

When the program is started, it tokenizes the input file specified when the application was started, as outlined in the Project 1 specification. Once it has tokenized the input file, it checks the token list’s syntax as specified in the C- Language specification (p491). It does this by passing the token set to a lexical analyzer which parses the tokens by using a series of rules as followed:

A-> B  
B->C B’  
C->E id ;  
E->int|void|float

\* This is not indiciative of the rule set that was used in the project. It’s just an example.

Whithin the Lexical Analyzer, each rule is represented by a function. The set of tokens that the analyzer uses is a global variable, as is the current token. The Analyzer pops the first token off of the token stack and calls A(). A() calls B() and so on until function E() is encountered. When E() is called and the current token is found to pass the rule of E (is it “int”, “void”, or “float”?), the token is “consumed” and the current token becomes the next token in the set. At this point, the code is turned into intermediate code by the magic of cigarettes and caffeine. The analyzer continues like this until all tokens are consumed, all of the intermediate code is generated and the analyzer is in an accepting state (a.k.a. we end up back at A() with nothing left to do).

# Sample Intermediate code Generation

### Sample Input

int x;

int y[3];

void sub2(int x, int y[]){

if(x==y[1]){

x=2;

while(x==2){

y[1]=2;

}

}

else{

x=3;

}

}

int sub(int x){

int y[4];

if(sub(x)>sub(3)){

if(x==3){

x =3;

}

}

else{

x=3;

}

x=3;

}

### Sample Output

//allocate global variables

1 alloc 4 x

2 mult 3 4 \_t1

3 alloc \_t1 y

//declare a function

4 func sub2 Void 2

5 params

//function params

6 alloc 4 x

7 alloc 4 y

//compare a value to an array index

//calculate displacement and displace y;

8 mult 1 4 \_t2

9 disp \_t2 y \_t3

//compare

10 cmpr x \_t3 \_t4

//go to if-code

11 breq \_t4 13

//go to else code

12 break 22

//assign x

13 assign 2 x

//loop while x==2

14 cmpr x 2 \_t5

//Go to code

15 breq \_t5 17

16 break 21

//assign a vale to an array index

17 mult 1 4 \_t6

18 disp \_t6 y \_t7

19 assign 2 \_t7

//go to top of loop

20 break 14

//skip else statement

21 break 23

//start else

22 assign 3 x

//end function

23 end func sub2

24 func sub Int 1

25 params

26 alloc 4 x

27 mult 4 4 \_t9

28 alloc \_t9 y

29 arg 3

30 call sub 1 \_t11

31 arg x

32 call sub 1 \_t10

33 cmpr \_t10 \_t11 \_t12

34 brgt \_t12 36

35 break 41

36 cmpr x 3 \_t13

37 breq \_t13 39

38 break 40

39 assign 3 x

40 break 42

41 assign 3 x

42 assign 3 x

43 end func sub