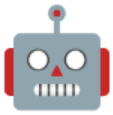

CSC 254 A3 README

Collaborators

Name	NetID
Kevin Tusiime	rtusiime
Enting Zhou	ezhou12

Description

In this project, we implemented a fully-functional compiler in OCaml that compiles programs written in our calculator language and converts them to low-level pidgin C or generates a relevant error.

```
----------------
Description of files

Source:
ecl.ml: the main program

Scripts:
clean.sh: bash script to swipe out generated executatbles
test.sh: do the following
1. compile ecl.ml to executable
2. use executable to translate examples test ecl program to c code
3. compile the translated c code
gen_ref.sh: bash script to generate c code from provided solution

Executables:
translate: the compiled code for our ecl.ml
ref: the sample solution provided

-----------------
How to test the program

The easiest way is to do the following:
1. put the test files into test_files
2. tweak the test.sh to incorporate new test_files
```

```
3. 'sh test.sh'

To test semantic check, simple
./translate< <Semantically broken ecl program>

and examine the output in terminal
```

Files

- **ecl.ml -** Source code for the compiler.
- **clean.sh -**
- **test.sh -**

How to build

Run the following command:

```
ocamlc -o ecl str.cma ecl.ml
```

where ecl is the name of the new file, **str.cma** is the string library were using and **ecl.ml** is the name of our compiler source code.

```
cycle1> ecl < testFile.ecl > testProgram.c
cycle1> gcc -o testProg testProgram.c
cycle1> ./testProg
```

where test is the text file that contains the program you'd like to parse.

Creating syntax tree

We recursively convert the generated parse tree (a list of either terminal or nonterminal nodes) to a syntax tree by walking down the parse tree using a collection of mutually recursive subroutines. Our syntax tree is structured as a list of statement and expression nodes

There are six main subroutines:

- `ast_ize_prog (p:parse_tree) : ast_sl` → This is the “root” call, to which we pass a parse tree, triggering it to call `ast_ize_stmt_list` since the parse tree is a list of statement lists.
- `ast_ize_stmt_list (sl:parse_tree) : ast_sl` → This will return either an empty list or the result of appending calls to `ast_ize_stmt` and `ast_ize_stmt_list`.
- `ast_ize_stmt (s:parse_tree) : ast_sl` → This will match against all nonterminal nodes of the parse tree with Statement (`S`) as a left-hand side and return syntax tree node comprising of the right-hand side of the particular production with S as it’s l.h.s. If these syntax tree nodes contain an expression, another call is made to `ast_ize_expr` within the construction of the node so as to creat a syntax tree node for the enclosed expression.
- `ast_ize_expr (e:parse_tree) : ast_e` → This will match against all nonterminal nodes of the parse tree with Condition (`C`), Expression (`E`), Term(`T`), or Factor(`F`) as their left-hand side and return a syntax tree comprising of the right-hand side of the particular production. If these syntax tree nodes contain an expression, or a factor_taill another call is made to `ast_ize_expr` or `ast_ize_expr_tail` respectively, within the construction of the node so as to create a syntax tree node for the enclosed expression/ expression_tail.
- `ast_ize_expr_tail (lhs:ast_e) (tail:parse_tree) : ast_e` → This will return the passed in expression syntax tree or will match against nonterminal nodes of the parse tree with `TT` , `FT` as their left-hand side and return syntax tree node comprising the right hand side of the particular production.
- `ast_ize_cond (c:parse_tree) : ast_e` → This will match against nonterminal nodes of the parse tree with `C` as their left-hand side and return a binary operation syntax tree node comprising the right hand side of the particular production(usually other expressions which then trigger calls to `ast_ize_expr`).

Translating the sytax tree

We translated syntax tree to equivalent pigdin C which we test for correctness by compiling it with a C compiler.

In the process of doing so, we check for undefined variables, redefinition, type clashes, and that scope rules are followed. In the case of scope rules, we ensure that each variable is visible from its declaration to the end of the innermost statement list in which

it is declared unless the variable has been redeclared in an inner statement list, of which the latter declaration will take precedence in inner statement lists over the outer declaration.

The bulk of the translation happens in the following subroutines:

- `translate_sl (sl:ast_sl) (st:symtab): symtab * string list * string list`
- `translate_read (id:string) (loc:row_col)(st:symtab): symtab * string list * string list`
- `translate_write (expr:ast_e) (st:symtab): symtab * string list * string list`
- `translate_assign (id:string) (rhs:ast_e) (vloc:row_col) (aloc:row_col) (st:symtab): symtab * string list * string list`
- `translate_if (c:ast_e) (sl:ast_sl) (st:symtab): symtab * string list * string list`
- `translate_while (c:ast_e) (sl:ast_sl) (st:symtab) symtab * string list * string list`
- `translate_expr (expr:ast_e) (st:symtab): symtab * string list * string list`
- `translate_ast (ast:ast_sl) : int * int * string * string`

each subroutine's return value is a tuple containing two integers and two strings. If the program represented by the AST is semantically correct, the first of the returned strings comprises equivalent pidgin C code, and the second string is empty, and the integers indicate the amount of needed memory and temporary space. If the AST has static semantic errors, the first string is empty and the second is a sequence of helpful error messages.

To aid in type checking, we created three helper functions

1. `get_expr_type (expr: ast_e) (st: symtab) : tp * string list * symtab`
   [line 953 in ecl]
   Which returns the type of a given AST node— if it's a nonterminal node which contains an inner expression, it the type of the expression. If it's a terminal node, it returns the type of the id by looking up the id in a symbol table.
2. `binary_check (lhs: ast_e) (rhs: ast_e) (op_loc: row_col) (st: symtab) (st: symtab): string list * tp * symtab`
   [line 992 in ecl]

Which checks whether the type of the expression enclosed on either side of the binary operators such as `*` , `+` , `-` , `<>`, `==`, `/`

3. `unary_check (expr: ast_e) (op_loc: row_col) (op_tp: tp) (st: symtab): string list * tp * symtab`
   [line 983 in ecl]
   Which returns the type of the expression enclosed within unary type operators such as `float()` and `trunc()`

4. `type_clash_assign_check (id: string) (id_loc: row_col) (rhs: ast_e) (assign_loc: row_col) (st: symtab): string list * symtab`
   [line 1007 in ecl]
   This assumes that input has already been checked with id_check and checks whether the expression type is the same as that of a declared variable.

Memory management

We implemented the temporaries management by adding extra fields in the symble table, that is it keep track of how many variables are used and how many temporaries are used. Note we did not implement the stack-based allocation so we never reuse any temporaries. Whenever the program declares a new variable, a new symbol table will be created with the variable inserted to the scope and an increased variable counter by 1. The same goes with temporaries. In handling the expression, we store the memory associated with the expression in the expression's text field inside the operand returned by the translate_expr procedure. That way it is easy to locate which memory address is associated with the expression.

Semantic Errors

We successfully caught the following semantic errors:

1. Use of an undeclared variable.
   a. Simple testcase

```
read real a;
write f;
```

b. More complicated testcase

```
read real a;
write a;
read int b;
write b;
if a<>1.0
    then
    a := a+1.0;
    a := a*a*a*2.001;
    end;
 a:=12343.34343343;
int v := trunc(f);
```

2. Redeclaration of a variable in the same scope.

```
read int a;
read int b;
read real c;
while a <> b do
    if a > b then
        int c := 3;
        a := a - b;
        if a > 0 then
            int q := a-c;
        end;
    end;
    real z := c - 0.003;
    if b > a then
        b := b - a;
    end;
    if a == b then
        write a;
    end;
end;
```

1. Non-integer provided to `trunc`.

```
int a := 4;
int b := trunc(a);
```

1. Non-real provided to `float`.

```
real b := float(3.003);
```

```
read real a;
write a;
read int b;
write b;
if a<>1.0
    then
    a := a+1.0;
    a := a*a*a*2.001;
    end;
 a:=12343.34343343;
int v := trunc(b);
```

2. Type clash in binary expression, comparison, or assignment.

```
int a := trunc(float(7)*3) + 3;
```

```
int a := 7;
int b := 8;
int v := float(a*b);
```

```
int a := 7;
int b := 8;
real v := float(a*b);
```

```
int b := 0.2;
```

Limitations to the program

The prorgam cannot reuse out of scope temporaries, we did not implement the stack based allocation. (we are in 254)