Introduction
The C Flat language is mostly a subset of the C language. Some of the core functionalities of C has been stripped: there is no preprocessor, no structs, no strings, not even pointers. It's goal is purely educational.

Originally Nico and Dan were working on two separate languages. The two projects merged, taking some features from each, and this is the resulting language.

The ltc and C Flat language proposals are included in appendix A.

C Flat Tutorial
C Flat is easy to use for any programmer familiar with a C-like language. The main differences from C are that there is only one type: the integer, variables don't have to be declared before use (but there are no global variables), and there are exceptions.

"Hello World" (outputs 1):
```c
main() {
    out(1);
}
```

Recursive Fibonacci:
```c
fib(n) {
    if (n < 3) return 1;
    else return fib(n-1) + fib(n-2);
}
main() {
    out(fib(in()));
}
```

Iterative Fibonacci:
```c
fib(n) {
    if (n < 1)
        throw 1;
    a = 1; // last fib #
    b = 1; // current fib #
    for (i = 3; i <= n; i++) {
        temp = b;
        b += a;
        a = temp;
    }
    return b;
}
main() {
    out(fib(in()));
}
```

Language Manual
The LRM is included in appendix B.
Project Plan

Processes

When we merged projects, we decided to start off by implementing as much of the ltc proposal as possible and adding features from the original c flat proposal at the end if we had time. The ltc proposal clearly laid out the subset of C which we would be implementing. Since we were basing the language off of an existing one, there wasn't very much planning that had to go into figuring out how the language would work as far as the users are concerned.

Our process for progressing through the project was to pick a feature that wasn't implemented and think through how exactly it should work and how it needed to be implemented. We sometimes had to compile some test C programs and look at the assembly generated or look up instructions in the Intel x86 manuals to learn exactly how something would work. Then we'd implement the feature and some tests for it. Sometimes we wrote the corresponding part of the LRM at that point and sometimes we filled it in later. This normally wouldn't be a great idea, but this language is small enough that it worked just fine.

We had an automated tester which would run a series of code snippets through the compiler, run them, and verify that the output (or lack thereof) was correct. The tester was an improved version of the tester Dan used last fall in PLT. Initially we were also running the test suite that professor Edwards supplied with microc, but we eventually migrated to only our tester. It was quicker and easier to write tests for this tester because they all go in one file. After we did any work on the compiler, we'd run a quick "make test" and be able to verify that everything still worked correctly.

Programming Style Guide

For the compiler, we stuck with the style already used in microc. More or less:

• Indentation: 2 spaces.
• Indentation level is increased when declaring a non trivial function (that is a function with at least one argument)
• When matching, each case should be on it's own line. It also increase the indentation level.
• Function names are in lower case with words separated by underscores.
• Structures: names and fields are lower case with words separated by underscores.
• Types: names lowercase, possible values first letter uppercase.
• Tester uses standard python style as outlined in PEP 8 - http://www.python.org/dev/peps/pep-0008/
• Tests: lower case variables and functions, 2 space indentation.
For example, this is a snippet from backend.ml:

```ocaml
let rec eval_expr_to_eax fdecl =
    function
    Literal(l) ->
        sprintf "mov eax, %d\n" l
    | Assignop(v, o, e) ->
        let assign_binop binop =
            eval_expr_to_eax fdecl (Assignop(v, Assign, Binop(Id(v), binop, e))) in
        (match o with
        Assign        -> eval_expr_to_eax fdecl e ^
            sprintf "mov [ebp+%d], eax\n" (id_to_offset fdecl v)
        | Add_assign    -> assign_binop Add
        | Sub_assign    -> assign_binop Sub
        | Mult_assign   -> assign_binop Mult
        | Div_assign    -> assign_binop Div
```

**Project Timeline**
- October: Proposals, automated tester.
- November: Merged projects, first assembly program generated (skeleton, functions, basic operators, I/O, if, for, and while), LRM started.
- December:
  - Week 1: Nothing.
  - Week 2: Many operators, proper argument evaluation, break, continue, exceptions.
  - Week 3: Static semantic analysis, compiler completed, LRM completed.

**Roles and Responsibilities**
We worked together on most aspects of the project. Nico implemented a number of language features on his own and Dan got the automated tester going.

**Software Development Environment**
We are both running a linux OS with its standard tools.

<table>
<thead>
<tr>
<th>Kate, Vim, and Nano</th>
<th>Source code editing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ocaml Tool Suite</td>
<td>Lexer, parser, static and semantic analysis, backend, top level compiler driver</td>
</tr>
<tr>
<td>Gcc</td>
<td>Compiling standard library (which is C), assembling output of the C Flat compiler, linking object files</td>
</tr>
<tr>
<td>Python</td>
<td>Automated tester</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>Report</td>
</tr>
<tr>
<td>Lyx</td>
<td>LRM</td>
</tr>
<tr>
<td>GNU Make</td>
<td>Project building</td>
</tr>
</tbody>
</table>
**Project Log**

We used git as a version control system. The project history is:

Nicolas Viennot 2008-12-18 10:47:49 -0500 style fix
Nicolas Viennot 2008-12-18 10:45:37 -0500 style fix
Nicolas Viennot 2008-12-18 10:39:08 -0500 style fix
Nicolas Viennot 2008-12-18 10:01:23 -0500 lrn updated
Nicolas Viennot 2008-12-18 08:37:28 -0500 removed dead code
Nicolas Viennot 2008-12-18 08:34:34 -0500 not pushing esp for exception
Nicolas Viennot 2008-12-17 21:44:26 -0500 added test for shifting negative number
Nicolas Viennot 2008-12-17 18:03:25 -0500 removed the goto keyword
Nicolas Viennot 2008-12-17 16:45:58 -0500 added if/else test (2)
Nicolas Viennot 2008-12-17 16:44:52 -0500 added if/else test
Daniel Benamy 2008-12-17 16:05:22 -0500 Merge branch 'work'
Daniel Benamy 2008-12-17 16:05:00 -0500 Renamed microc to cflat.
Nicolas Viennot 2008-12-17 16:01:14 -0500 Assign is not an assignop
Daniel Benamy 2008-12-17 15:54:58 -0500 Added a couple of tests for exceptions.
Nicolas Viennot 2008-12-17 15:48:23 -0500 precedence change for < >
Daniel Benamy 2008-12-17 15:47:16 -0500 Print compiler errors to stderr.
Nicolas Viennot 2008-12-17 15:35:01 -0500 unclosed comment raise exception
Nicolas Viennot 2008-12-16 19:51:18 -0500 cleanup
Nicolas Viennot 2008-12-16 12:37:58 -0500 asm test pretty
Nicolas Viennot 2008-12-16 12:33:02 -0500 small asm change
Nicolas Viennot 2008-12-16 08:37:29 -0500 added .PHONY : test/clean in Makefile
test: testing for local variable discovery and duplicates.
sast: forgot a variable check
Nicolas Viennot 2008-12-15 20:09:35 -0500 cleanup
Nicolas Viennot 2008-12-15 20:07:04 -0500 sast: variables are added through the context struct
Nicolas Viennot 2008-12-15 19:44:13 -0500 cleanup
Nicolas Viennot 2008-12-15 14:42:54 -0500 cleanup
Nicolas Viennot 2008-12-15 14:14:57 -0500 removed test directory
Nicolas Viennot 2008-12-15 14:08:45 -0500 removing old tests
Nicolas Viennot 2008-12-15 14:06:09 -0500 more tests
Nicolas Viennot 2008-12-15 12:10:25 -0500 cleanup
Nicolas Viennot 2008-12-15 12:10:15 -0500 added test when mixing same function name/variable name
Nicolas Viennot 2008-12-15 12:09:25 -0500 catch exception on syntax error
Nicolas Viennot 2008-12-15 10:41:14 -0500 ast printer removed
Nicolas Viennot 2008-12-15 10:38:39 -0500 local variables are now initialized to 0
Nicolas Viennot 2008-12-15 10:07:50 -0500 SAST added, local variables doesn't need to be declared anymore
Nicolas Viennot 2008-12-14 20:38:40 -0500 "uncaught exception" message added
Nicolas Viennot 2008-12-14 14:28:08 -0500 cleanup
Nicolas Viennot 2008-12-14 14:14:41 -0500 reversing args in function call is done is assembly
Nicolas Viennot 2008-12-14 13:37:41 -0500 precedence test added
Nicolas Viennot 2008-12-14 12:26:43 -0500 operator precedence fixed
Nicolas Viennot 2008-12-14 09:40:40 -0500 cleanup
Nicolas Viennot 2008-12-14 09:13:02 -0500 sign tests added for arithmetic binops
Nicolas Viennot 2008-12-14 09:12:45 -0500 using movzx instead of mov eax, 0
Nicolas Viennot 2008-12-13 16:33:28 -0500 cleanup
Nicolas Viennot 2008-12-13 15:14:11 -0500 exception implemented
Nicolas Viennot 2008-12-13 10:03:40 -0500 basic try/catch/throw implementation
Nicolas Viennot 2008-12-13 09:12:36 -0500 Operators implemented and tested
Nicolas Viennot 2008-12-13 07:43:56 -0500 comment scanner fixed
Nicolas Viennot 2008-12-13 07:32:01 -0500 arguments of a function are evaluated from left to right
Nicolas Viennot 2008-12-13 07:00:41 -0500 cleanup
Nicolas Viennot 2008-12-12 14:49:58 -0500 all operators added, exception added (not finished !!)
Nicolas Viennot 2008-12-12 09:20:40 -0500 added multiline comments
Nicolas Viennot 2008-12-12 09:12:31 -0500 reverted out() -> outputs \
and test programs are piped to xargs
Nicolas Viennot 2008-12-12 00:08:43 -0500 Added test for double variable declarationsWhitespace fix.
Nicolas Viennot 2008-12-12 00:06:55 -0500 Added lrm.
Added break and continue keywords.
Added tests.
Nicolas Viennot 2008-12-11 23:07:00 -0500 Removed interpreter.
Nicolas Viennot 2008-12-11 22:04:34 -0500 Implemented proper labels.Implemented in().
Changed out() to not add newline and added outln() and ln().
Fixed make test.
Nicolas Viennot 2008-12-11 20:25:43 -0500 removed test from make clean
Nicolas Viennot 2008-12-02 00:35:04 -0500 new test case, cleanup
Nicolas Viennot 2008-12-02 00:21:38 -0500 new test
<table>
<thead>
<tr>
<th>Author</th>
<th>Date/Time</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nicolas Viennot</td>
<td>2008-12-02 00:18:13 -0500</td>
<td>global removed</td>
</tr>
<tr>
<td>Nicolas Viennot</td>
<td>2008-12-02 00:15:15 -0500</td>
<td>global tests removed</td>
</tr>
<tr>
<td>Nicolas Viennot</td>
<td>2008-12-02 00:12:32 -0500</td>
<td>okey whatever</td>
</tr>
<tr>
<td>Nicolas Viennot</td>
<td>2008-12-02 00:11:25 -0500</td>
<td>microc deleted</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-11-22 02:11:46 -0500</td>
<td>Added ltc and microc.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-11-22 02:10:36 -0500</td>
<td>Started on a backend for microc to produce x86 assembly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Put some skeleton file in cflat/, moved testing code to cflat/test, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>added the very beginnings of a test file for c flat.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-11-22 02:06:49 -0500</td>
<td>Fix crash when no executable is created.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-10-16 02:49:59 -0400</td>
<td>Added hw1 dir.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-10-16 01:24:07 -0400</td>
<td>Give nice error if no test file given.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-10-16 00:38:26 -0400</td>
<td>Changed test markers to ..., added support for comments before</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compilercommand, and cleanup.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-10-16 00:24:50 -0400</td>
<td>Various fixes. Tester works.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-10-16 00:09:09 -0400</td>
<td>Cleanup.</td>
</tr>
<tr>
<td>Daniel Benamy</td>
<td>2008-10-16 00:07:48 -0400</td>
<td>Importing compiler-tester.py in progress.</td>
</tr>
</tbody>
</table>
Our translator receives a C Flat program and outputs the x86 assembly code:

The translator first tokenizes the character stream, parses it to generate an AST. Then the tree is semantically checked to produce a SAST which goes through the backend to produce assembly code.

A few notes about the implementation:

- The semantic analyzer does the local variables discovery so that the backend knows in advance the stack size for local variables.
- Function calls: we follow the C convention that is: arguments are pushed in reverse order, and the caller cleans the stack. The register eax is used for return values.
- Exceptions: we use a linked list that is built on the stack. An element is added to the list when the program enters a try block. When an exception is thrown, it checks if the list is empty; if yes uncaught_exception() is called, if not it passes control to the catch block.
- Temporary values: during a complex expression evaluation, we use the stack to store temporary results.

Nico adapted the parser and the scanner from the microc code and implemented the semantic analyzer. Both Dan and Nico implemented the Backend.
**Test Plan**  
Sample compilations

**Recursive Fibonacci:**  
C Flat:
```
c fib(n) {
    if (n < 0) throw -1;
    if (n < 3) return 1;
    else return fib(n-1) + fib(n-2);
}
c main() {
    try {
        out(fib(in()));
    } catch(ex) {
        out(ex);
    }
}
```

Generated assembly:
```
.intel_syntax noprefix
.text
.globl main
.type main, @function
main:
    push ebp
    mov ebp, esp
    xor eax, eax
    push eax
    push ecx
    push edx
    push ebp
    push offset .L1
    push dword ptr __exception_ptr
    mov __exception_ptr, esp
    call in
    add esp, 0
    push eax
    call fib
    add esp, 4
    push eax
    call out
    add esp, 4
    mov eax, __exception_ptr
    mov eax, [eax]
    mov __exception_ptr, eax
    add esp, 12
    jmp .L2
.L1:
    mov [ebp+-4], edx
    mov eax, [ebp+-4]
    push eax
    call out
    add esp, 4
.L2:
.L0:
```
pop   edx
pop   ecx
mov   esp, ebp
pop   ebp
ret
.globl fib
.type fib, @function
fib:
push ebp
mov   ebp, esp
xor   eax, eax
push ecx
push edx
mov   eax, [ebp+8]
push eax
mov   eax, 0
pop   ecx
xchg eax, ecx
cmp   eax, ecx
setl  al
movzx eax, al
test   eax, eax
jz     .L4
mov   eax, 1
neg    eax
mov   edx, eax
mov   ecx, [__exception_ptr]
test   ecx, ecx
jnz    .L6
push edx
call   __uncaught_exception
.L6:
mov   eax, [__exception_ptr]
mov   eax, [eax]
mov   [__exception_ptr], eax
lea    esp, [ecx+12]
mov   ebp, [ecx+8]
jmp   [ecx+4]
jmp   .L5
.L4:
.L5:
mov   eax, [ebp+8]
push eax
mov   eax, 3
pop    ecx
xchg   eax, ecx
cmp    eax, ecx
setl   al
movzx   eax, al
test   eax, eax
jz     .L7
add    esp, 0
mov   eax, 0
jmp   .L3
jmp   .L8
.L7:
add    esp, 0
mov   eax, [ebp+8]
push eax
mov eax, 1
pop ecx
xchg eax, ecx
push eax
call fib
add esp, 4
push eax
mov eax, [ebp+8]
push eax
mov eax, 2
pop ecx
xchg eax, ecx
sub eax, ecx
push eax
call fib
add esp, 4
pop ecx
xchg eax, ecx
add eax, ecx
jmp .L3
.L8:
.L3:
pop edx
pop ecx
mov esp, ebp
pop ebp
ret
.ident "C Flat compiler 0.1"
GCD:
C Flat:
```c
gcd(a, b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
main() {
    a = in();
    b = in();
    out(gcd(a, b));
}
```

Generated assembly:
```
.intel_syntax noprefix
.text
.globl main
.type main, @function
main:
    push ebp
    mov ebp, esp
    xor eax, eax
    push eax
    push ecx
    push edx
    push ebp
    push offset .L1
    push dword ptr [__exception_ptr]
    mov [__exception_ptr], esp
    call in
    add esp, 0
    push eax
    call fib
    add esp, 4
    push eax
    call out
    add esp, 4
    mov eax, [__exception_ptr]
    mov eax, [eax]
    mov [__exception_ptr], eax
    add esp, 12
    jmp .L2
.L1:
    mov [ebp+-4], edx
    mov eax, [ebp+-4]
    push eax
    call out
    add esp, 4
.L2:
.L0:
    pop edx
    pop ecx
    mov esp, ebp
    pop ebp
```
ret
.globl fib
.type fib, @function
fib:
push ebp
mov ebp, esp
xor eax, eax
push ecx
push edx
mov eax, [ebp+8]
push eax
mov eax, 0
pop ecx
xchg eax, ecx
cmp eax, ecx
setl al
movzx eax, al
test eax, eax
jz .L4
mov eax, 1
neg eax
mov edx, eax
mov ecx, [__exception_ptr]
test ecx, ecx
jnz .L6
push edx
call __uncaught_exception
.L6:
mov eax, [__exception_ptr]
mov eax, [eax]
mov [__exception_ptr], eax
lea esp, [ecx+12]
mov ebp, [ecx+8]
jmp [ecx+4]
jmp .L5
.L4:
.L5:
mov eax, [ebp+8]
push eax
mov eax, 3
pop ecx
xchg eax, ecx
cmp eax, ecx
setl al
movzx eax, al
test eax, eax
jz .L7
add esp, 0
mov eax, 1
jmp .L3
jmp .L8
.L7:
add esp, 0
mov eax, [ebp+8]
push eax
mov eax, 1
pop ecx
xchg eax, ecx
sub eax, ecx
push eax
call fib
add esp, 4
push eax
mov eax, [ebp+8]
push eax
mov eax, 2
pop ecx
xchg eax, ecx
sub eax, ecx
push eax
call fib
add esp, 4
pop ecx
xchg eax, ecx
add eax, ecx
jmp .L3
.L8:
.L3:
pop edx
pop ecx
mov esp, ebp
pop ebp
ret
.ident "C Flat compiler 0.1"

Test suite
Our tester is a small python program which parses a text file with a very simple format, runs the test cases it finds, and verifies that the correct result is produced. See appendix C for the tester code (tester.py).

Test cases
Each feature of the compiler is tested. During the development, when we added a feature, we wrote a test case for it to validate the implementation. At least one test case is written for each specification of the LRM. See appendix C for test cases (test-cflat.txt).

Automation
A simple “make test” rebuilds the compiler if needed and executes all test cases. Making the test process very accessible is important since we use a test-driven development

Division of Testing Work
Dan wrote the automated tester. We wrote tests as we were adding features so we wrote a number of tests together, mainly checking things in microc. Then when Nico implemented all the additional operators, exceptions, and static semantic analysis, he wrote most of the tests for those.
Lessons Learned

Dan learned that Ocaml is pretty cool. Functional programming takes getting used to, and this project was a bunch of getting-used-to-ness.

Nico learned that Ocaml is a very nice language and that compilers are not magical anymore.

We would strongly encourage future teams to do automated testing and add test cases for each feature before or at roughly the same time as the feature it's testing. In addition to serving as a verification of functionality and preventing regressions, the act of writing code in the new language helps with figuring out how things should work and can bring up odd cases that might otherwise go unnoticed.

We recommend that teams reuse our tester since it lowers the effort required to add tests to about as low as possible.
Appendix A - Language proposals
Cb (C flat)
Daniel Benamy

Introduction
This is the language proposal for a new language, Cb (pronounced "see flat"). It is not intended to be a complete language reference manual, although it may morph into one.

Cb is a toy language which I’m using to learn how to write a compiler in OCaml. It contains a basic subset of C with some additions, including functions as first class types, and exceptions.

Target
Will compile to C.

Whitespace
Will use python-style nesting via indentation. The start of a block will end with a ':.' and each level of nesting is indicated by indenting 4 spaces. Newlines are statement separators. Any number of spaces, including zero, are allowed around identifiers and punctuation.

Comments
C and C++ style comments allowed.

// I am a single line comment.
/* I am a multi-line comment. */
Multi-line comments end with the first */ found.

Control Flow
Control flow is achieved using functions, if statements, while statements, and exceptions.

Control starts in the main function which must exist. It doesn't take any arguments and must return an int.

    int main():
    return 0

Types
int, float (implemented as a double), char, bool, func.

Func is a multi-type. The types that the function takes and returns must be specified.

    func{int; int, int} f // Declares a variable f which is a function that takes two ints and returns an int.

Operators
This table lists the available operators and what types they take as arguments. All operators return the same type passed to them except for the equality and inequality operators (<, <=, >, >=, ==, !>) which return bool.

| + | - | * | / | & (bitwise and) | | || (logical or) | | < | <= | > | >= | == | != | ! |
|---|---|---|---|---------------|---|----------------|---|---|---|---|---|---|---|
| int | X | * | X | X | X | X | X | X | X | X | X | X | X |
**Type System**
Variables are statically typed. Once a variable is assigned a type, it can not be changed.

**Typecasting**
Int can be implicitly cast to floats when assigning to variables or passing to operators or functions. No other automatic typecasting or promotion is done.

Anything can be converted to a string using the built in string() function.

**Binding**
Cb is statically bound. The compiler figures out what all identifiers mean at compile time.

**Variable Creation and Use**
Variables don’t need to be declared before use, but can be. This is only recommended for cases where the type can’t be determined automatically. They can’t be used until assigned to. Assignment is done with ‘=’. An assignment statement can have one or more values per side although the same number must be on each side. If more than one is given, they are comma separated.

There are no global or static variables.

**Functions**
Functions are called using the () syntax. A function may be called by name, or a func variable may be called.

```c
add_one(int i):
    return i + 1
```

```c
int main():
    add_one(5)
    f = add_one
    f(4)
    return 0
```

Functions take 0 or more arguments and return 0 or more values.

```c
foo(int a):
    ...
```

or

```c
int, char bar():
    return 5, 'a'
```

Arguments are passed with copy semantics.

There is no function overloading.

Functions must be declared before being used. Recursion is allowed.
Functions can only be declared at the top level of the source code. I.e., not within other functions.

**Exceptions**
Functions can throw exceptions which causes cascading "returns" of the exception until a catch is found. Exceptions are ints. If an exception is not caught, the program terminates. If an exception is caught, the int thrown is assigned to the variable specified in the catch statement.

```python
foo(int x):
    try:
        bar(x)
        return 0
    catch e:
        return 1
```

```python
bar(int x):
    throw 1
```

**Namespace**
There is one namespace for all functions, variables, and types.

**I/O**
Done with
- `out(char c)`, `err(char c)`
and
- `char in()`

**Milestones**
1. Grammar for all v 1 features.
2. Automated test framework (grab from last PLT project).
4. I/O.
5. Function declaration and calling.
6. Func type. Calling functions via func variables.
7. Exceptions.
8. The initial implementation will not include arrays or strings. If time permits (hah), I will add them later.

**Arrays (v 2)**
Array elements are accessed with []. Arrays keep track of their size and accesses are bounds checked.
Arrays are reference counted.

What do they contain?
memory management
declaration
manipulation

**Strings (v 2)**
The string type is syntactic sugar for arrays of chars.

Implicit casting to string?
COMS W4115 - Project Proposal

ltc
less than C
Introduction
The ltc is a very small subset of the C language. The project consists of creating a compiler written in ocaml that translate ltc code into assembly.

Overview
The standard GNU tools as and ld are used as assembler and linker.

Language features
The ltc syntax is quite similar to C: statements end with ";", blocks are surrounded with "{ }". the language has very limited features: arrays and pointers are not supported and there are no variable types (only integer). The major differences between ltc and C are described as follows:

Variables
- variables can only be 32 bits signed integers
- variables don't need to be declared
- global/static variables are not allowed
- variables are accessible in the function scope

Operators
- only the aritmetic, comparison, bitwise and assignment operators are available
- operator precedence is the same as in C.

Functions
- functions can be recursive
- arguments are passed by value
- functions always return an integer and the return value is implicitly given by the result of the last statement
- inline functions are not available
**Control**
- the do..while construction is not available
- the switch keyword is not available
- labels, goto, if/else, while and for loops, behave like C

**Comments**
- /* this is a comment */, they can be nested
- // this is also a comment

**External calls**
External functions can be called by including the right .o object files while linking. No “extern” declaration is needed, the linker will complain if the function doesn't exist.
To communicate with the external world, two functions are provided in the “library”: `input()` and `output()`.

**Generated Assembly**
The generated assembly is not optimized at all. Temporary values are stored on the stack (not in registers), like every other variables which is very inefficient.

**Code sample**
```c
pow(a, b)
{
    if (b == 0)
        1; /* no return keyword */
    else
        a * pow(a, b-1); /* recursion */
}
main()
{
    a = b = input(); /* read from stdin */
    b += 2;
    for (i = 1; i <= 5; i++) {
        c = pow(a, i);
        output(2 * (c + b)); /* print to stdout */
    }
    1;
}
```
Appendix B - LRM
1 Introduction
The C Flat language is mostly a subset of the C language. Some of the core functionalities of C has been stripped: there is no preprocessor, no structs, no strings, not even pointers. It’s goal is purely educational. Originally Nico and Dan were working on two separate languages. The two projects merged, taking some features from each, and this is the resulting language. This document is inspired by the C Reference Manual by Dennis Ritchie.

2 Lexical conventions

2.1 Whitespace
A tab, a space or a new line is a whitespace. At least one of these characters is required to separate adjacent identifiers, constants, and certain operator-pairs.

2.2 Comments
There are two ways to place comments: // introduces a comment which ends with a end of line. /* also introduces a comment which ends with */, they can be nested. A // inside a /* */ comment is ignored.

2.3 Identifiers
An identifier is a sequence of letters and digits. The first character must be alphabetic. The underscore counts as alphabetic. An identifier is case sensitive.

2.4 Keywords
The following identifiers are reserved for use as keywords, and may not be used otherwise:
    return break continue if else for while try catch throw

2.5 Constants
There is only one kind of constant: a 32 bits signed integer. Such a constant is a sequence of digits represented in its decimal form.

3 Expressions
An expression evaluates to a 32 bits signed integer. The precedence of operators is described in the syntax summary.

3.1 identifier
An identifier evaluates to the value of the corresponding variable.

3.2 literal
A decimal number is an expression.
3.3 ( expression )
A parenthesized expression evaluates to the parenthesized expression.

3.4 identifier ( expression-list opt )
A function call is an expression. The arguments are optional and separated with a comma. They are evaluated from left to right before the call (applicative order). The value returned by the function is the value the callee returns with a return statement.

3.5 ~expression
The result is the negative of the expression.

3.6 +expression
The result is the expression itself.

3.7 !expression
The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero.

3.8 ~expression
The ~ operator yields the one’s complement of its operand.

3.9 identifier++
The referred variable is incremented when evaluated. The expression evaluates to the value of the variable before the increment.
Note that the statement “{a = 0; b = 0; b = a++ + a++; }” sets the value of a to 2, and b to 1.

3.10 identifier--
The referred variable is decremented when evaluated. The expression evaluates to the value of the variable before the decrement.

3.11 ++identifier
The referred variable is incremented when evaluated. The expression evaluates to the value of the variable after the increment.

3.12 --identifier
The referred variable is decremented when evaluated. The expression evaluates to the value of the variable after the decrement.

3.13 expression * expression
The binary * operator indicates multiplication.

3.14 expression / expression
The binary / operator indicates division.
3.15 expression \% expression
The binary \% operator yields the remainder from the division of the first expression by the second. The remainder has the same sign as the dividend.

3.16 expression + expression
The result is the sum of the expressions.

3.17 expression - expression
The result is the difference of the expressions.

3.18 expression >> expression
expression << expression
The value of the right hand side operand should be non-negative and less than 32, if not the result is undefined. The value of “E1 >> E2” is E1 arithmetically right-shifted by E2 bit positions. Vacated bits are filled by a copy of the sign bit of the first expression. The value of “E1 << E2” is R1 left-shifted by E2 bit positions. Vacated bits are 0-filled.

3.19 expression < expression
expression > expression
expression <= expression
expression >= expression
The operators < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

3.20 expression == expression
expression != expression
The operators == (equal to) and the != (not equal to) yield 0 if the specified relation is false, 1 if it is true.

3.21 expression & expression
The & operator yield the bitwise and function of the operands.

3.22 expression ^ expression
The ^ operator yield the bitwise exclusive or function of the operands.

3.23 expression | expression
The | operator yield the bitwise inclusive or function of the operands.

3.24 expression \&\& expression
The \&\& operator returns 1 if both operands are non-zero, 0 otherwise. Both operands are always evaluated.

3.25 expression || expression
The || operator returns 1 if either of its operands is non-zero, 0 otherwise. Both operands are always evaluated.
3.26  identifier = expression
The value of the referred variable is replaced by the value of the expression.

3.27  identifier += expression
      identifier -= expression
      identifier *= expression
      identifier /= expression
      identifier %= expression
      identifier >>= expression
      identifier <<= expression
      identifier &= expression
      identifier ^= expression
      identifier |= expression

An expression of the form “id op= expr” is equivalent to “id = id op expr”.

4  Statements
Statements are executed in sequence.

4.1  Expression statement
Most statement are expression statements, which have the form
      expression ;

4.2  Compound statement
So that several statements can be used where one is expected, the compound statement is provided:
      compound-statement:
            { statement-list_opt }
      statement-list:
            statement
            statement statement-list

4.3  Conditional statement
The two forms of the conditional statement are
      if ( expression ) statement
      if ( expression ) statement else statement

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case
the second substatement is executed if the expression is 0. As usual the “else” ambiguity is resolved by connecting
an else with the last encountered elseless if.
4.4 While statement
The while statement has the form

    while ( expression ) statement

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

4.5 For statement
The for statement has the form

    for ( expression-1_opt ; expression-2_opt ; expression-3_opt ) statement

This statement is equivalent to

    expression-1;
    while ( expression-2 ) {
        statement
        expression-3;
    }

Any or all the expression may be dropped. A missing expression-2 makes the implied while clause equivalent to “while(1)”. Other missing expressions are simply dropped from the expansion above.

4.6 Break statement
The statement

    break;

causes termination of the smallest enclosing while or for statement; control passes to the statement following the terminated statement.

4.7 Continue statement
The statement

    continue;

causes control to pass to the loop-continuation portion of the smallest enclosing while or for statement; that is to the end of the loop. In case of a for loop of the form “for(e1;e2;e3) {...}”, e3 is evaluated before checking e2.

4.8 Return statement
A function returns to its caller by means of the return statement

    return expression ;

The value of the expression is returned to the caller of the function.

4.9 Null statement
The null statement has the form

    ;

A null statement is useful to supply a null body to a looping statement such as while.
4.10 Try-catch statement

The two forms of the try-catch statement are

```plaintext
try { statement-list_opt }catch ( identifier ) { statement-list_opt }
```

```plaintext
try { statement-list_opt }catch { statement-list_opt }
```

The statements enclosed in the try block are executed until an exception is thrown. In case no exception is thrown, the statements enclosed in the catch block are not executed. The first form of the try-catch statement allows to assign the value of the exception to a variable. Try-catch statement dynamically nest across function calls.

4.11 Throw statement

The throw statement has the form

```plaintext
throw expression ;
```

Throwing an exception causes control to pass to the catch block of the nearest dynamically-enclosing try-catch statement. If none is found, it causes the program to terminate with an error. The given expression is the value of the thrown exception.

5 Program definition

A ltc program consists of a sequence of function definition.

```plaintext
program:
function-definition
function-definition program

function-definition:

identifier ( parameter-list_opt ) { statement-list_opt }

parameter-list:

identifier
identifier , parameter-list
```

the same identifier cannot be used more than once in the parameter list. Within the same program, A function cannot be defined twice (name wise).

All functions return a integer value. A function can return to the caller without an explicit return statement, in this case the return value is undefined.

A simple example of a complete function definition:

```plaintext
max (a, b, c) {
if (a > b) m = a; else m = b;
if (m > c) return m; else return c;
}
```
6 Scope rules

There are no global variables, but only local variables which are statically binded. The scope of a local variable is the whole function where the variable is used. The scope of function parameters is the whole function.

Function scope is the entire program.

7 Declarations

Variables don’t need to be declared, they are initialized to 0.

A function call can be made whether or not the function actually exists, the program will simply not link if a call to a non-existing function is made.

8 Namespace

Variables and function use different namespaces. Therefore such a function is correct: “\( f() \{ f=1; \text{ return } f; \} \)”.

9 Syntax Summary

9.1 Expressions

expression:

identifier

literal

(expression)

identifier (expression-list_opt)

-expression

+expression

!expression

~expression

++identifier

--identifier

identifier++

identifier--

expression binop expression

identifier asgnop expression

expression-list:

expression

expression , expression-list

The unary operators \(- + ! ~\) have higher priority than binary operator.

Binary operators all group left to right and have priority decreasing as indicated:

binop:

* / %
Assignment operator all have the same priority, and all group right to left.

\[
\text{asgnop:} \quad = \quad += \quad -= \quad *= \quad /= \quad %= \quad >>= \quad <<= \quad &= \quad \&\& \quad |\|
\]

9.2 Statements

statement:

\[
\text{expression} \; ; \\
\{ \text{statement-list}_{\text{opt}} \} \\
\text{if} \; (\text{expression}) \; \text{statement} \\
\text{if} \; (\text{expression}) \; \text{statement} \; \text{else} \; \text{statement} \\
\text{while} \; (\text{expression}) \; \text{statement} \\
\text{for} \; (\text{expression}_{\text{opt}}; \text{expression}_{\text{opt}}; \text{expression}_{\text{opt}}) \; \text{statement} \\
\text{break}; \\
\text{continue}; \\
\text{return} \; \text{expression}; \\
\text{try} \; \{ \text{statement-list}_{\text{opt}} \} \text{catch} \; \{ \text{statement-list}_{\text{opt}} \} \\
\text{try} \; \{ \text{statement-list}_{\text{opt}} \} \text{catch} \; (\text{identifier}) \; \{ \text{statement-list}_{\text{opt}} \} \\
\text{throw} \; \text{expression}; \\
\]

statement-list:

\[
\text{statement} \\
\text{statement} \; \text{statement-list} \\
\]
9.3 Program definition

program:

    function-definition
    function-definition program

function-definition:

    identifier ( parameter-list_opt ) { statement-list_opt }

parameter-list:

    identifier
    identifier , parameter-list
Appendix C - source
OBJS = parser.cmo scanner.cmo backend.cmo sast.cmo cflat.cmo

CFLAGS="-m32"

cflat : $(OBJS) lib.o
  ocamlc -o cflat $(OBJS)

.PHONY: test
test : cflat tester.py test-cflat.txt
  ./tester.py test-cflat.txt

scanner.ml : scanner.mll
  ocamllex scanner.mll

parser.ml parser.mli : parser.mly
  ocamlyacc parser.mly

%.cmo : %.ml
  ocamlc -c $<

%.cmi : %.mli
  ocamlc -c $<

.PHONY : clean
clean :
  rm -f cflat parser.ml parser.mli scanner.ml testall.log *.cmo *.cmi *.o *.s test

# Generated by ocamldep *.ml *.mli
backend.cmo: ast.cmi
backend.cmix: ast.cmi
cflat.cmo: scanner.cmo sast.cmo parser.cmi backend.cmo
cflat.cmix: scanner.cmix sast.cmix parser.cmix backend.cmix
parser.cmo: ast.cmi parser.cmi
parser.cmix: ast.cmi parser.cmix
sast.cmo: ast.cmi
sast.cmix: ast.cmi
scanner.cmo: parser.cmi
scanner.cmix: parser.cmix
parser.cmi: ast.cmi
let newline    = '\n' | "\r\n"
let whitespace = [ ' ' | '\t' ] | newline
let digit      = ['0'-'9']
let integer    = digit+
let alpha      = ['a'-'z' 'A'-'Z']
let alphanum   = alpha | digit
let identifier = alpha alphanum*

rule token = parse
    whitespace { token lexbuf }
    | "//" { comment_double_slash lexbuf }
    | "/*" { comment_slash_star 0 lexbuf }

(* arithmetic operators *)
    | "++" { INC }
    | "--" { DEC }
    | "+=" { MINUS_ASSIGN }
    | "+=" { PLUS_ASSIGN }
    | "-=" { TIMES_ASSIGN }
    | "/=" { DIVIDE_ASSIGN }
    | "+=" { TIMES_ASSIGN }
    | "+=" { PLUS }  
    | "+=" { TIMES }  
    | "+=" { DIVIDE }  
    | "+=" { MODULO }  

(* must be before the "|" and "&" *)
    | "&&" { AND }  
    | "||" { OR }  

(* bitwise operators *)
    | "<<=" { LSHIFT_ASSIGN }
    | ">>=" { RSHIFT_ASSIGN }
    | "&=" { BW_AND_ASSIGN }
    | "|=" { BW_OR_ASSIGN }
    | "^=" { BW_XOR_ASSIGN }
    | "<<" { LSHIFT }  
    | ">>" { RSHIFT }  
    | "~" { BW_NOT }  
    | "&" { BW_AND }  
    | "|" { BW_OR }  
    | "+" { BW_XOR }  

(* logic operators *)
(* done before
    | "&&" { AND }
    | "||" { OR }
*)
    | "<=" { LEQ }  
    | ">=" { GEO }  
    | "!=" { NEQ }  

scanner.mll
/* punctuation */
| "="       { ASSIGN } |
| '('       { LPAREN } |
| ')'       { RPAREN } |
| '{'       { LBRACE } |
| '}'       { RBRACE } |
| ';'       { SEMI } |
| ','       { COMMA } |

/* keywords */
| "for"     { FOR } |
| "while"   { WHILE } |
| "if"      { IF } |
| "else"    { ELSE } |
| "return"  { RETURN } |
| "break"   { BREAK } |
| "continue" { CONTINUE } |
| "try"     { TRY } |
| "catch"   { CATCH } |
| "throw"   { THROW } |
| integer as lit        { LITERAL(int_of_string lit) } |
| identifier as id      { ID(id) } |

| eof        { EOF } |
| _ as char  { raise (Failure("illegal character " ^ Char.escaped char)) } |

and comment_slash_star level = parse
"/*"      { if level = 0 then token lexbuf 
             else comment_slash_star (level-1) lexbuf } |
"*/"      { comment_slash_star (level+1) lexbuf } |
| eof       { raise (Failure("Comment not closed")) } |
| _         { comment_slash_star level lexbuf } |

and comment_double_slash = parse
newline   { token lexbuf } |
| _         { comment_double_slash lexbuf}
%{ open Ast %}

%token INC DEC MINUS ASSIGN PLUS ASSIGN TIMES_ASSIGN DIVIDE_ASSIGN MODULO_ASSIGN
%token MINUS ASSIGN PLUS TIMES DIVIDE MODULO
%token LSHIFT_ASSIGN RSHIFT_ASSIGN BW_AND_ASSIGN BW_OR_ASSIGN BW_XOR_ASSIGN
%token LSHIFT RSHIFT ASSIGN BW_NOT ASSIGN BW_AND_ASSIGN BW_OR_ASSIGN BW_XOR_ASSIGN
%token LEQ GEQ NEQ EQ NOT AND OR LT GT
%token ASSIGN LPAREN RPAREN LBRACE RBRACE SEMI COMMA
%token FOR WHILE IF ELSE RETURN BREAK CONTINUE TRY CATCH THROW
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right BW_AND_ASSIGN BW_XOR_ASSIGN BW_OR_ASSIGN LSHIFT_ASSIGN RSHIFT_ASSIGN
%left TIMES_ASSIGN DIVIDE_ASSIGN MODULO_ASSIGN PLUS_ASSIGN MINUS_ASSIGN ASSIGN
%left OR
%left AND
%left BW_OR
%left BW_XOR
%left BW_AND
%left EQ NEQ
%left GT GEQ LT LEQ
%left LSHIFT RSHIFT
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%nonassoc NOT BW_NOT U_PLUS U_MINUS

%start program
%type <Ast.program> program

%%

program: /* nothing */ { [] } | program fdecl { $2 :: $1 }
fdecl: ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
       { { _fname = $1;
          _formals = $3;
          _body = List.rev $6 } }
formals_opt: /* nothing */ { [] } | formal_list { List.rev $1 }
formal_list: ID { [$1] } | formal_list COMMA ID { $3 :: $1 }
stmt_list: /* nothing */ { [] }
stmt_list stmt { $2 :: $1 }

stmt:
  expr SEMI { Expr($1) }
  SEMI { Expr(Noexpr) }
  RETURN expr SEMI { Return($2) }
  LBRACE expr SEMI { Block(List.rev $2) }
  IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
  FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) }
  WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  BREAK SEMI { Break }
  CONTINUE SEMI { Continue }
  TRY LBRACE stmt_list RBRACE CATCH LBRACE stmt_list RBRACE { Try_catch(Block(List.rev $3), "", Block(List.rev $7)) }
  TRY LBRACE stmt_list RBRACE CATCH LPAREN ID RPAREN LBRACE stmt_list RBRACE { Try_catch(Block(List.rev $3), $7, Block(List.rev $10)) }
  THROW expr SEMI { Throw($2) }

expr_opt:
  /* nothing */ { Noexpr }
  expr { $1 }

expr:
  LITERAL { Literal($1) }
  ID { Id($1) }
  expr OR expr { Binop($1, Or, $3) }
  expr AND expr { Binop($1, And, $3) }
  expr BW_OR expr { Binop($1, Bw_or, $3) }
  expr BW AND expr { Binop($1, Bw_and, $3) }
  expr BW XOR expr { Binop($1, Bw_xor, $3) }
  expr LSHIFT expr { Binop($1, Lshift, $3) }
  expr RSHIFT expr { Binop($1, Rshift, $3) }
  expr PLUS expr { Binop($1, Add, $3) }
  expr MINUS expr { Binop($1, Sub, $3) }
  expr TIMES expr { Binop($1, Mult, $3) }
  expr DIVIDE expr { Binop($1, Div, $3) }
  expr MODULO expr { Binop($1, Modulo, $3) }
  expr EQ expr { Binop($1, Equal, $3) }
  expr NEQ expr { Binop($1, Neq, $3) }
  expr LT expr { Binop($1, Less, $3) }
  expr LEQ expr { Binop($1, Leq, $3) }
  expr GT expr { Binop($1, Greater, $3) }
  expr GEQ expr { Binop($1, Geq, $3) }

  NOT expr { Unop(Not, $2) }
  BW_NOT expr { Unop(Bw_not, $2) }
  PLUS expr %prec U_PLUS { Unop(Plus, $2) }
  MINUS expr %prec U_MINUS { Unop(Minus, $2) }
  INC ID { Incop(Pre_inc, $2) }
  DEC ID { Incop(Pre_dec, $2) }
  ID INC { Incop(Post_inc, $1) }
  ID DEC { Incop(Post_dec, $1) }

  ID BW_AND_ASSIGN expr { Assignop($1, Bw_and_assign, $3) }
  ID BW OR ASSIGN expr { Assignop($1, Bw_or_assign, $3) }
ID BW_XOR_ASSIGN expr { Assignop($1, Bw_xor_assign, $3) }
ID LSHIFT_ASSIGN expr { Assignop($1, Lshift_assign, $3) }
ID RSHIFT_ASSIGN expr { Assignop($1, Rshift_assign, $3) }
ID TIMES_ASSIGN expr { Assignop($1, Mult_assign, $3) }
ID DIVIDE_ASSIGN expr { Assignop($1, Div_assign, $3) }
ID MODULO_ASSIGN expr { Assignop($1, Modulo_assign, $3) }
ID PLUS_ASSIGN expr { Assignop($1, Add_assign, $3) }
ID MINUS_ASSIGN expr { Assignop($1, Sub_assign, $3) }
ID ASSIGN expr { Assignop($1, Assign, $3) }
ID LPAREN actuals_opt RPAREN { Call($1, $3) }
ID LPAREN expr RPAREN { $2 }

actualls_opt:
/* nothing */ { [] } 
| actualls_list { List.rev $1 }

actualls_list:
expr { [$1] } 
| actualls_list COMMA expr { $3 :: $1 }
type binop = Add | Sub | Mult | Div | Modulo | Or | And | Bw_or | Bw_and | Bw_xor | Lshift | Rshift | Equal | Neq | Less | Leq | Greater | Geq

type assignop = Assign | Add_assign | Sub_assign | Mult_assign | Div_assign | Modulo_assign | Bw_or_assign | Bw_and_assign | Bw_xor_assign | Lshift_assign | Rshift_assign

type unop = Not | Bw_not | Plus | Minus

type incop = Pre_inc | Post_inc | Pre_dec | Post_dec

type expr =
  Literal of int
  | Id of string
  | Unop of unop * expr
  | Incop of incop * string
  | Binop of expr * binop * expr
  | Assignop of string * assignop * expr
  | Call of string * expr list
  | Noexpr

type stmt =
  Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Break
  | Continue
  | Try_catch of stmt * string * stmt
  | Throw of expr

type func_decl = {
  _fname : string;
  _formals : string list;
  _body : stmt list;
}

type program = func_decl list

type func_decl_detail = {
  fname : string;
  formals : string list;
  locals : string list;
  body : stmt list;
}

type program_detail = func_decl_detail list
sast.ml

open Ast
open Printf

type context = {
  in_loop : bool;
  variables : string list ref;
}

(* returns l1 - l2 *)
let rec diff_list l1 = function
  | [] -> l1
  | hd2 :: tl2 ->
      let rec diff_hd2 = function
        | [] -> []
        | hd1 :: tl1 ->
            if hd1 = hd2 then diff_hd2 tl1
            else hd1 :: diff_hd2 tl1
      in
      diff_list (diff_hd2 l1) tl2

(* add v to the context.variables list if not in the list *)
let add_variable context v =
  let rec add_unique_v = function
    | [] -> [v]
    | hd :: tl ->
        if hd = v then hd :: tl
        else hd :: add_unique_v tl
  in
  context.variables := add_unique_v !(context.variables)

let rec check_expr fdecl context = function
  | Literal(_) -> ()
  | Id(v) -> add_variable context v
  | Unop(_, e) -> check_expr fdecl context e
  | Incop(_, v) -> add_variable context v
  | Binop(e1, _, e2) -> check_expr fdecl context e1;
      check_expr fdecl context e2
  | Assignop(v, _, e) -> add_variable context v;
      check_expr fdecl context e
  | Call(_, el) -> List.iter (check_expr fdecl context) el
  | Noexpr -> ()

let rec check_stmt fdecl context = function
  | Block(sl) -> List.iter (check_stmt fdecl context) sl
  | Expr(e) -> check_expr fdecl context e
  | Return(e) -> check_expr fdecl context e
  | If(e, s1, s2) -> check_expr fdecl context e;
      check_stmt fdecl context s1;
      check_stmt fdecl context s2
  | For(e1, e2, e3, s) ->
      let context' = { context with in_loop = true } in
      check_expr fdecl context' e1;
      check_expr fdecl context' e2;
      check_expr fdecl context' e3;
      check_stmt fdecl context' s
  | While(e, s) -> check_stmt fdecl context (For(Noexpr, e, Noexpr, s))
  | Break -> if not context.in_loop then
raise (Failure("break keyword used outside a loop"))
| Continue             -> if not context.in_loop then
|                        raise (Failure("continue keyword used outside a loop"))
| Try_catch(s1, v, s2)  -> check_stmt fdecl context s1;
|                        add_variable context v;
|                        check_stmt fdecl context s2
| Throw(e)             -> check_expr fdecl context e

(* check a func_decl and returns a func_decl_detail *)
let check_func fdecl =
(* first check that each formal is only declared once *)
let rec check_formal_unique formal_list formal =
  (match formal_list with
   [] -> [formal]
  | hd :: tl ->
    if hd = formal then
      raise (Failure("formal " ^ formal ^ " is declared more than once" ^
          " in function " ^ fdecl._fname))
    else
      hd :: check_formal_unique tl formal)
let _ = List.fold_left check_formal_unique [] fdecl._formals in

let context = {
  in_loop = false;
  variables = ref []
} in
check_stmt fdecl context (Block(fdecl._body));
{ fname = fdecl._fname;
  formals = fdecl._formals;
  locals = diff_list !(context.variables) fdecl._formals;
  body = fdecl._body }

(* check a program and returns a program_detail *)
let check_program funcs =
(* first we check that a function is only declared once *)
let rec check_funcs_unique fname_list fdecl =
  (match fname_list with
   [] -> [fdecl._fname]
  | hd :: tl ->
    if hd = fdecl._fname then
      raise (Failure("function " ^ fdecl._fname ^
          " is declared more than once"))
    else
      hd :: check_funcs_unique tl fdecl)
let _ = List.fold_left check_funcs_unique [] funcs in
List.map check_func funcs
open Ast
open Printf

type context = {
    label_count : int ref;
    break_label : string option;
    continue_label : string option;
    return_label : string option;
    function_try_level : int;
    loop_try_level : int;
}

let get_new_label context =
    let l = !context.label_count in
    context.label_count := l + 1;
    ".L" ^ (string_of_int l)

let get = function
    Some(x) -> x
  | None -> ""

let rec index_of item n = function
    [] -> -1
  | hd::tl -> if hd = item then n else (index_of item (n+1) tl)

let id_to_offset fdecl id =
    let n = index_of id 0 fdecl.formals in
    if n >= 0 then
        4 * (n+2)
    else
        let n = index_of id 0 fdecl.locals in
        if n >= 0 then
            -4 * (n+1)
        else
            raise (Failure("undefined identifier " ^ id))

(*
an exception looks like this:
struct exception {
    struct exception *next;
    void *catch_address;
    int old_ebp;
};
*)

let exception_context_size = 3*4

let stack_exception catch_label =
    "push ebp\n" ^
    sprintf "push offset %s\n" catch_label ^
    "push dword ptr __exception_ptr\n" ^
    "mov __exception_ptr, esp\n"

let unstack_exception n =
let rec unwind_exception = function
  0 -> ""
| n -> "mov eax, [__exception_ptr]\n" ^
   "mov eax, [eax]\n" ^
   "mov [__exception_ptr], eax\n" ^
   unwind_exception (n-1)

let rec eval_expr_to_eax fdecl = function
  Literal(l) -> sprintf "mov eax, %d\n" l
| Id(s) -> sprintf "mov eax, [ebp+%d]\n" (id_to_offset fdecl s)
| Unop(o, e) ->
  eval_expr_to_eax fdecl e ^
  (match o with
   Not -> "test  eax, eax\n" ^
         "setz  al\n" ^
         "movzx eax, al\n"
   | Bw_not -> "not  eax\n"
   | Plus -> ""
   | Minus -> "neg  eax\n")
| Incop(o, v) ->
  let asm = function
    Pre_inc | Post_inc ->
    sprintf "inc dword ptr [ebp+%d]\n" (id_to_offset fdecl v)
  | Pre_dec | Post_dec ->
    sprintf "dec dword ptr [ebp+%d]\n" (id_to_offset fdecl v) in
  (match o with
   Pre_inc | Pre_dec -> asm o ^ eval_expr_to_eax fdecl (Id v)
   | Post_inc | Post_dec -> eval_expr_to_eax fdecl (Id v) ^ asm o)
| Binop(e1, o, e2) ->
  eval_expr_to_eax fdecl e1 ^
  "push eax\n" ^
  eval_expr_to_eax fdecl e2 ^
  "pop  ecx\n" ^
  "xchg eax, ecx\n" ^
  (* eax contains e1, ecx contains e2 *)
  (match o with
   Equal | Neq | Less | Leq | Greater | Geq ->
   "cmp eax, ecx\n" | _ -> "") ^
  (match o with
   Add -> "add  eax, ecx\n" | Sub -> "sub  eax, ecx\n" | Mult -> "imul  eax, ecx\n" |
   Div -> "cdq\n" ^
   "idiv  ecx\n" | Modulo -> "cdq\n" ^
   "idiv  ecx\n" ^
   "mov  eax, edx\n"
| Or      -> "or eax, ecx\n" ^  
|          | "setnz al\n" ^  
| And     -> "test eax, eax\n" ^  
|          | "setnz al\n" ^  
|          | "test ecx, ecx\n" ^  
|          | "setnz cl\n" ^  
|          | "and al, cl\n"  
| Bw_or   -> "or eax, ecx\n"  
| Bw_and  -> "and eax, ecx\n"  
| Bw_xor  -> "xor eax, ecx\n"  
| Lshift  -> "sal eax, cl\n"  
| Rshift  -> "sar eax, cl\n"  
| Equal   -> "sete al\n"  
| Neq     -> "setne al\n"  
| Less    -> "setl al\n"  
| Leq     -> "setle al\n"  
| Greater -> "setg al\n"  
| Geq     -> "setge al\n"

(match o with  
 Or | And | Equal | Neq | Less | Leq | Greater | Geq ->  
     | _ -> "")

| Assignop(v, o, e) ->  
| let assign_binop binop =  
| eval_expr_to_eax fdecl (Assignop(v, Assign, Binop(Id(v), binop, e))) in  
(match o with  
 Assign       -> eval_expr_to_eax fdecl e ^  
|              | sprintf "mov [ebp+%d], eax\n" (id_to_offset fdecl v)  
| Add_assign   -> assign_binop Add  
| Sub_assign   -> assign_binop Sub  
| Mult_assign  -> assign_binop Mult  
| Div_assign   -> assign_binop Div  
| Modulo_assign-> assign_binop Modulo  
| Bw_or_assign -> assign_binop Bw_or  
| Bw_and_assign-> assign_binop Bw_and  
| Bw_xor_assign-> assign_binop Bw_xor  
| Lshift_assign-> assign_binop Lshift  
| Rshift_assign-> assign_binop Rshift)  

| Call(f, el) ->  
| let push_func_args =  
| let prepare_arg e =  
| eval_expr_to_eax fdecl e ^  
| "push eax\n" in  
String.concat "" (List.map prepare_arg el) ^  
(let rec reverse_all_args i j =  
if i < j then  
swap_two_args i j ^  
reverse_all_args (i+1) (j-1)  
else "" in  
reverse_all_args 0 (List.length el - 1) in  
push_func_args ^
let rec string_of_stmt context fdecl = function
  Block(stmts) -> String.concat "" (List.map (string_of_stmt context fdecl) stmts)
| Expr(expr) -> eval_expr_to_eax fdecl expr
| Return(expr) -> unwind_exception context.function_try_level ^
  unstack_exception context.function_try_level ^
  eval_expr_to_eax fdecl expr ^
  sprintf "jmp %s\n" (get context.return_label)
| If(e, s1, s2) ->
  let else_label = get_new_label context
  and exit_if_label = get_new_label context in
  eval_expr_to_eax fdecl e ^
  "test eax, eax\n" ^
  sprintf "jz   %s\n" else_label ^
  string_of_stmt context fdecl s1 ^
  sprintf "jmp %s\n" exit_if_label ^
  sprintf "%s:\n" else_label ^
  string_of_stmt context fdecl s2 ^
  sprintf "%s:\n" exit_if_label
| For(e1, e2, e3, s) ->
  let loop_begin_label = get_new_label context
  and loop_label = get_new_label context in
  let context' = { context
                     with continue_label = Some loop_label;
                     break_label = Some loop_exit_label;
                     loop_try_level = 0
                  } in
  eval_expr_to_eax fdecl e1 ^
  sprintf "jmp %s\n" loop_begin_label ^
  sprintf "%s:\n" loop_label ^
  eval_expr_to_eax fdecl e3 ^
  sprintf "%s:\n" loop_exit_label ^
  (match e2 with
    | _ -> eval_expr_to_eax fdecl e2 ^
      "test eax, eax\n" ^
      sprintf "jz   %s\n" loop_exit_label) ^
  string_of_stmt context' fdecl s ^
  sprintf "jmp %s\n" loop_label ^
  sprintf "%s:\n" loop_exit_label
| While(e, s) ->
  string_of_stmt context fdecl (For(Noexpr, e, Noexpr, s))
| Break ->
  unwind_exception context.loop_try_level ^
  unstack_exception context.loop_try_level ^
  sprintf "jmp %s\n" (get context.break_label)
unwind_exception context.loop_try_level ^
unstack_exception context.loop_try_level ^
sprintf "jmp %s\n" (get context.continue_label)

Try_catch(s1, v, s2) ->
  let catch_label = get_new_label context
  and exit_label = get_new_label context in
  let context' = { context
    with
      function_try_level = context.function_try_level + 1;
      loop_try_level = context.loop_try_level + 1}
  in
  stack_exception catch_label ^
  string_of_stmt context' fdecl s1 ^
  unwind_exception 1 ^
  unstack_exception 1 ^
  sprintf "jmp %s\n" exit_label ^
  sprintf "%s:\n" catch_label ^
  (match v with
     "" -> ""
  | _  -> sprintf "mov [ebp+%d], edx\n" (id_to_offset fdecl v)) ^
  string_of_stmt context fdecl s2 ^
  sprintf "%s:\n" exit_label

Throw(e) ->
  let caught_exception = get_new_label context in
  eval_expr_to_eax fdecl e ^
    "mov edx, eax\n" ^
    "mov ecx, [_exception_ptr]\n" ^
    "test ecx, ecx\n" ^
    sprintf "jnz %s\n" caught_exception ^
    "push edx\n" ^
    "call __uncaught_exception\n" ^
    sprintf "%s:\n" caught_exception ^
  unwind_exception 1 ^
    "lea esp, [ecx+12]\n" (* exception is unstacked *)
    "mov ebp, [ecx+8]\n" ^
    "jmp [ecx+4]\n"

let string_of_fdecl context fdecl =
  let context' = { context
    with
      return_label = Some (get_new_label context) }
  in
  sprintf ".globl %s\n" fdecl.fname ^
  sprintf ".type %s, @function\n" fdecl.fname ^
  sprintf "%s:\n" fdecl.fname ^
    (* creating frame *)
    "push ebp\n" ^
    "mov ebp, esp\n" ^
    (* instead of "sub esp, 4*num_locals", we "push 0" num_locals times,
       this way, the local variables are cleared on the fly *)
    "xor eax, eax\n" ^
  String.concat "\n" (List.map (fun _ -> "push eax\n") fdecl.locals) ^
  "push ecx\n" ^
  "push edx\n" ^
  string_of_stmt context' fdecl (Block(fdecl.body)) ^
  sprintf "%s:\n" (get context'.return_label) ^
    "pop edx\n" ^
    "pop ecx\n" ^
    "mov esp, ebp\n" ^
    "pop ebp\n" ^
let generate_asm funcs =
  let context = { label_count = ref 0;
                 continue_label = None;
                 break_label = None;
                 return_label = None;
                 function_try_level = 0;
                 loop_try_level = 0 } in
  "intel_syntax noprefix\n" ^
  "text\n" ^
  String.concat "" (List.map (string_of_fdecl context) funcs) ^
  "ident \"C Flat compiler 0.1\"\n"
let _ =
  try
    let lexbuf = Lexing.from_channel stdin in
    let program = Parser.program Scanner.token lexbuf in
    let program_detail = Sast.check_program program in
    print_string (Backend.generate_asm program_detail); exit 0
  with
    Failure(s) -> prerr_endline ("Error: " ^ s); exit 1
    | Parsing.Parse_error -> prerr_endline ("Syntax error"); exit 1
lib.c

#include <stdio.h>
#include <stdlib.h>

#define asmlinkage __attribute__((regparm(0)))
asmlinkage int in(void) {
    int i;
    scanf("%d", &i);
    return i;
}
asmlinkage void out(int val) {
    printf("%d
", val);
}
asmlinkage void __uncaught_exception(int ex) {
    printf("uncaught exception: %d
", ex);
    exit(1);
}

void *__exception_ptr;
def compile_and_run(compiler, code):
    """Returns (compile status, output).
    compile status is 'OK' or 'BAD'.
    If compile fails, output has compiler output. If compile succeeds, it has
    the output of the program run.
    """
    proc = Popen4(compiler)
    proc.tochild.write(code)
    proc.tochild.close()
    if proc.wait() != 0:
        return ('BAD', proc.fromchild.read())
    proc = Popen4('./test | xargs')
    return ('OK', proc.fromchild.read().strip())

def print_indented(message):
    for line in message.splitlines():
        print "   ", line

def run_test(compiler, code, correct_result):
    """Returns True if the test passes, otherwise False."""
    status, output = compile_and_run(compiler, code)
    if correct_result == 'BAD':
        if status == 'BAD':
            print "PASS"
            return True
        elif status == 'OK':
            print "FAIL: Bad code compiled. Code:",
            print_indented(code)
            return False
        elif correct_result == 'OK':
            if status == 'BAD':
                print "FAIL: Good code didn't compile. Code:",
                print_indented(code)
                print "Compiler output:"
                print_indented(output)
                return False
            elif status == 'OK':
                print "PASS"
                return True
        else:
            if status == 'BAD':
                print "FAIL: Good code didn't compile. Code:",
                print_indented(code)
                print "Compiler output:"
                print_indented(output)
                return False
            elif correct_result == output:
print "PASS"
return True
else:
    print "FAIL: Incorrect output from execution. Code:"
    print_indented(code)
    print "Executed code output:"
    print_indented(output)
    print "Correct output:"
    print_indented(correct_result)
    return False

if len(sys.argv) < 2:
    print "You must specify a test file."
    exit(1)
test_file_name = sys.argv[1]
test_file = open(test_file_name)
print "Loading test file '%s'." % test_file_name
compiler = '#'
while compiler.startswith('#'):
    compiler = test_file.readline().strip()
print "Using compile command '%s'." % compiler
test_count = 0
pass_count = 0
code = ""
for line in test_file:
    if line == "...
    code = ""
    elif line.startswith("...
    test_count += 1
    correct_result = line.strip("\n")[4:]
    if (run_test(compiler, code, correct_result)):
        pass_count += 1
    else:
        code += line
test_file.close()
print "%d / %d tests passed." % (pass_count, test_count)
test-cflat.txt

# The first line that doesn't start with a '#' should be the compiler command.  
# It should read source code from stdin and produce an executable named  
# 'test'.  
bash -c './cflat > test.s && gcc -m32 -c test.s -o test.o && gcc -m32 test.o 
lib.o -o test')

After the compiler command, anything not within a test case (surrounded by ...)  
is ignored.
The closing ... should be followed by a space and then the desired result of the 
test:  
* OK if the code should compile,  
* BAD if the code shouldn't compile, or  
* Any other single line string which the code, when run, should produce.   
Whitespace may be trimmed from the front or back.

*** Compiling ***
...
main() {
}
... OK

...
main() {
    bad
}
... BAD

*** Comments ***
...
main() { }
/*
... BAD

main() {
    out(1);
    /* out(2); /* out(3); */ // */
    /* garbage */
    out(4); // out(5); garbage
    out(6);
}
... 1 4 6

*** Variables ***
...
f(a, b) {
    out(a);
    out(b);
}
main() {
    out(a); /* a should be initialized to 0 */
a = 1;
b = a;
out(b);
out(a+b);
f(2, 3);
}
... 0 1 2 2 3
...
/* variable can have the same name as a function */
f(f) {
    out(f);
}
main() {
    f = 2;
    f(f);
} ...
...
/* local variable discovery */
dummy(a) { }
a() { v; }
b() { !v; }
c() { v++; }
d() { v=0; }
e() { 0+v; }
f() { v+=0; }
g() { v=0; }
h() { dummy(v); }
i() { { v; } }
j() { return v; }
k() { if (v); }
l() { for(v;;); }
m() { for(;v;); }
n() { for(;v); }
o() { while (v); }
p() { try {} catch(v) {} }
q() { throw v; }
main() { }
... OK

*** Operators correctness ***
...
main() {
    /* Unop */
    out(-3);
    out(+ 4);
    out(!0);
    out(!2);
    out(!2);
    out(~10);
    out(~10);
}
... -3 -4 1 0 1 -11 10

main() {
    /* increment and decrement */
main() {
    /* Arithmetic binops */
    out(3+1);
    out(3+-1);
    out(-2+-2);
    out(3-1);
    out(3- -1);
    out(-1- -2);
    out(3*2);
    out(-2*3);
    out(-1*-1);
    out(12/4);
    out(6/-2);
    out(-5/-5);
    out(10%4);
    out(10%-4);
    out(-10%4);
    out(-10%-4);
}

main() {
    /* Bitwise binops */
    out(3<<2);
    out(-1<<1);
    out(12>>2);
    out(-1>>1);
    out(1|4);
    out(3&5);
    out(3^5);
}

main() {
    /* Assign binops */
    a = 0; a += 2; out(a);
    a = 0; a -= 2; out(a);
    a = 2; a *= 3; out(a);
    a = 6; a /= 2; out(a);
    a = 7; a %= 4; out(a);
}
a = 3; a <<= 2; out(a);
a = 12; a >>= 2; out(a);
a = 1; a |= 4; out(a);
a = 3; a &= 5; out(a);
a = 3; a ^= 5; out(a);
}
... 2 -2 6 3 3 12 3 5 1 6

main() {
/* Logic binops */
out(-2>=-1);
out(0>=-1);
out(0>=0);
out(1>=0);
out(-2>=-1);
out(0>=-1);
out(0>=0);
out(1>=0);
out(-1<=-2);
out(-2<=0);
out(1<=1);
out(0<=1);
out(-1<=-2);
out(-2<=0);
out(1<=1);
out(0<=1);
out(1==1);
out(1==0);
out(1!=1);
out(1!=0);
out(0&&1);
out(1&&0);
out(1&&3);
out(0&&0);
out(0||0);
out(1||0);
out(0||1);
out(1||3);
}
... 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1

*** Operator precedence ***
...
op(expr, wanted, not_wanted) {
    out((expr == wanted) && (wanted != not_wanted));
}
main() {
/* left assoc test */
out(2) * out(3) * out(4);
/* precedence test */
op(~2*3, (~2)*3, ~(2*3));
op(1+2*3, 1+(2*3), (1+2)*3);
op(1<<2+3, 1<<(2+3), (1<<2)+3);
op(1<2+3, 1<(2+3), (1<2)+3);
op(1==2<1, 1==(2<1), (1==2)<1);
op(2&2==2, 2&(2==2), (2&2)==2);
op(1^2&3, 1^(2&2), (1^2)&2);
op(3|2^3, 3|(2^2), (3|2)^2);
op(0&&2|1, 0&&(2|1), (0&&2)|1);
op(1||2&0, 1||(2&0), (1||2)&0);

/* right assoc test */
a = 1; b = 2; c = 3;
a = b = c;
out(a); out(b); out(c);
}
... 2 3 4 1 1 1 1 1 1 1 1 1 3 3 3

*** Function test ***
...
/* should compile, number of argument are not checked */
f(a,b,c) { }
main() { f(a); }
... OK
...
/* duplicate function */
f() {}
f() {}
main() {}
... BAD
...
/* duplicate formals */
f(a,b,a) {}
main() {}
... BAD
...
/* Function call evaluation order */
f(a, b, c) { out(a); out(b); out(c); }
g(a, b, c) {}
main() {
    f(1, 2, 3);
g(out(4), out(5), out(6));
}
... 1 2 3 4 5 6
...
/* Recursive call test */
fib(x) {
    if (x < 3) return 1;
    return fib(x-1) + fib(x-2);
}
main() {
```c
out(fib(10));
}
...

...
main() {  
    f(); /* will fail at linking stage */
}
...
BAD

*** if/else ***
...
main() {
    if (1) out(1);
    else out(2);
    if (0) out(3);
    else out(4);
    if (0) out(5);
    else if (1) out(6);
    else out(7);
    if (1)
        if (0) out(8);
        else out(9);
    else out(10);
    if (1) out(11);
    if (0) out(12);
    else out(13);
}
...
1 4 6 9 11 13

*** Simple loop test ***
...
main() {
    sum = 0;
    for (i = 0; i <= 10; i++)
        sum += i;
    out(sum);
}
...
55

...
func(a) {
    return a-1;
}
main() {
    a = 10;
    while(func(a))
        a = func(a);
    out(a);
}
...
1
/* empty condition */
main() {
    for (; ; ) break;
    for (; ; i++) {
        if (i == 3)
            break;
        out(i);
    }
    ... 0 1 2
}

main() {
    while () { }
} ... BAD

*** Nested for/while ***
main() {
    for (i = 0; i < 2; i++) {
        while (0) { }
        for (j = 0; j < 2; j++)
            out (i + j);
    }
} ... 0 1 1 2

*** break/continue tests ***
main() {
    if (1) {
        break;
    }
} ... BAD

main() {
    continue;
} ... BAD

/* multi level break/continue */
main() {
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            if (j == 1)
                continue;
            else if (j == 3)
                break;
            out(i);
            out(j);
        }
        if (i == 2)
break;

... 0 0 0 2 1 0 1 2 2 0 2 2

*** Exceptions ***
... /* nested try/catch across functions */
g() {
  throw 4;
}
f() {
  try {
    g();
  } catch (b) {
    out(b);
    throw b-1;
  }
}
main() {
  out(1);
  try {
    f();
    out(2);
  } catch (a) {
    out(a);
  }
  out(5);
}
... 1 4 3 5

... /* exceptions in recursive functions caught outside */
f(x) {
  if (x == 0) {
    throw -1;
  } else {
    out(x);
    f(x-1);
  }
}
main() {
  try {
    f(3);
  } catch (e) {
    out(e);
  }
}
... 3 2 1 -1

... /* exceptions in recursive functions caught inside */
f(x) {
  try {
    if (x == 0) {
      throw -1;
    } else {

```java
out(x);
  f(x-1);
}
} catch(e) {
  out(e);
}
}
main() {
  f(3);
}
...
3 2 1 -1
...

/* continue/break in a loop */
main() {
  for (i=0; i < 5; i++) {
    try {
      try {
        if (i == 4)
          throw 2;
      } catch {
        break;
      }
      if (i == 2)
        continue;
    } catch { }
    out(i);
  }
  out(-1);
}
...
0 1 3 -1
...

/* return within a try */
f() {
  try {
    try {
      return 0;
    } catch { }
  }
}
main() {
  try {
    f();
    throw 1;
  } catch (a) {
    out(a);
  }
  out(2);
}
...
1 2
...
/* not catching an exception */
main() {
  throw 1;
}
... uncaught exception: 1
```