Compilers: K-assignment
Oversættere, DIKU, 2011-2012

Martin Jørgensen

20. januar 2012

Indhold

1 Introduction

I have implemented a new feature in a compiler for the language "100". This feature is the ability to handle exceptions. This is done with the `throw, try` and `catch` statements. In the assignment a grammar for the statements was given to describe the form of the statements. This report will document and explain the changes and expansion of the code for the compiler.

2 Lexer

The changes to the lexer was quite simple. In order for the lexer to recognise the keywords they were added to the keyword function. The three words added was:

- `throw`
- `try`
- `catch`

Lastly in order to properly recognise the whole pattern, a regular expression that picks up colons (:) was added.

3 Parser

I implemented the following rules in the Parser, according to the grammer given in the assignment text.

Stat → `throw` *Exp*
Stat → `try` *Stats* `catch` **id** : *Stat*

This was done by first adding `throw, try` and `catch` as tokens in the top of the Parser file. `catch` was also added as right associative along with the existing `else`. Finally `COLON` was added as a token at the top of the file to allow it to be tokenized proberly.

In the *Stat* definition; `throw` was added as a clone of `return`, since the share the exact same properties. The `try/catch` *Stat* was added as described in the grammar.

In order to pass the types correctly along, they had to be added to 100.sml as descriped in the next chapter.

4 S100

In this file the statements were added under the *Stat* datatype. Again the `throw` statement was made as a clone of the `return` statement.

The second statement, now added as `TryCatch` was added with these parameters: `Stat list * Sid * Stat * pos` as the grammer described. I briefly considered replacing the `Stat list` with a single `Stat` since it could just contain a `Block` but ended on deciding on the list simply because it was more like the grammar described.

5 Typechecker

In the typechecker I needed only to change two functions, as described below.

5.1 checkStat

5.1.1 throw

For the `throw` statement all that needed to be checked was if the *Exp* returns an integer, if it does, a unit is returned, else an *Error* is raised.

5.1.2 TryCatch

For the `TryCatch` steatement I more or less copied the `Block` check, with the following corrections:

- The alternative vtable was created by adding a tuble consisting of the *sid*'s name and the *Int* type.
- Since the `TryCatch` statement has both a *Stat* and a *Stat list*, instead of only running one thing, I created a tuble with the *Stat* being checked and then using List.app all the stats in the stat list gets checked.

5.2 checkReturn

Adding `throw` to checkReturn was easy, since it is essentially `return` it was just added as a clone of that. Adding `TryCatch` was a mix of `IfElse` and `Block` since the *Stats* is similar to a `block`, but the *Stat* is similar to `IfElse`.

5.3 Unhandled exception

I added an function called `unhandled` that have both argument and return type *Int*. This will be the function that handles unhandled exceptions by writing the predefined string and then an integer given by the triggering `throw` statement.

6 Compiler

6.1 Overall

In the `compileStat` call in `compileFun` I added the arguments needed for the changes described below. The catch label was set to `unhandled`, so if a statement does not have another catch-label assigned it will go to unhandled. I also created a new variable for execption values. This actual value will not be used but it will be used later.

6.2 compileStat

I added 2 arguments to this function *catchLabel* and *catchVar*. The label is used to provide throws a point to jump to. *catchVar* works in a similar way, but provides a variable to save the expression from `throw` in.

6.2.1 Throw

The code for throw is again pretty much a copy of the code for return, with the minor change that it will jump to the *catchLabel* and it stores its result in register 4 since that is were the `unhandled` function takes its arguments. It uses the *catchVar* to save the result of the expression. The final code looks like this:
`expCode @ [Mips.OR("4","0",catchVar), Mips.J catchLabel]`

6.2.2 TryCatch

`TryCatch` create 3 labels, one for the start, the catch and the end. It then adds a new variable to an alternate symboltable that will be used in the catch-stat. The try-stat list is compiled with the original f- and vtable and with the catch-entry-label as *catchLabel*. The catch-stat will be compiled with the altered vtable and the *catchLabel* along with the added exception variable. The final code looks like this:
`[Mips.LABEL l1] @ stat1Code @ [Mips.J l3, Mips.LABEL l2] @ stat2Code @ [Mips.LABEL l3]`

6.3 Unhandled Exception

The only predefined function I created was the one that catches unhandled exceptions. It is extremely simple albeit very long because of the operations requred to load the string unto the heap. First we reserve space on the stack to save the RA and the argument (exception number). It then reserves space on the heap for the string. It then proceeds to load chars, byte by byte unto the heap. Once loaded it loads a 0-byte to the heap and jump and links to putstring. When it returns, it loads the argument (exception number) from the stack and jump and links putint which puts the int after the string. After this it frees ups the stack space and terminates using a syscall. This function is also added to the static ftable used to compile functions.

7 Test and correctness

7.1 Testing with the provided programs

We had a couple of programs given in the assignment with input and expected output, this section describes the tests using those programs.

7.1.1 Error file testing

We were given 15 .100 programs that should give errors at compile-time. Furthermore for my assignment there were 2 more error programs, one that should not be able to compile and one that should but would always return uncaught exception 42.

For quicktesting with the 15 original error programs I used a bash-script that ran them all.[1] After reading the output I could conclude that all the programs returned the expected output.

There is one test program that do not present the written output but still returns *correct* output: Error03.100 is a program without main function. The compiler however returns "Unknown function getchar" which is also correct. After inspecting error03.100 I found out that it does indeed call a function called getchar which does not exist. The reason why this is reported before/ instead of the lack of main function is because `checkProg` in the typechecker checks for existence of the main function as the very last thing before finishing. Had error03.100 not called the nonexisting function, the lack of main function would be discovered. This was tested simply by removing the line that called `getchar`.

The error programs for my assignment was also tested and yielded the correct results.

7.1.2 Testing with working programs

My test procedure from the other given programs (that were not for the other assignments) was more manual since they don't share a name.

1. To test compiler with program.100 (assuming a linux shell):
2. `./C100 program`
3. If an expected output excists: `cat program.out`
4. If program.in excists: `cat program.in | java -jar Mars.jar program.asm`
5. If program.in does not excist: skip the cat part.
6. Compare outputs

Using this method I tested the compiler with all the programs. They all worked, however *except-find* and *except-queens* did not work. This is because I did not have time to implement cross-function catches. This error is described later.

[1] test.sh: I did not write this script myself, it was used in our group assignment.

7.2 Testing with own programs

I wrote 3 test programs called *mytest01-3*. Each of these programs highlight a specific feature or bug.

- Program #1:
  This program simply demonstrates that the uncaught exception works. It essentially does the same as *error-except02* but is shorter to prevent unintentional bugs. It takes an integer as input, if this integer is 0 it throws "Uncaught exception: 0" else it returns 1.
- Program #2:
  This program demonstrates that nesting a try/catch statement in another try/catch statement still works. The nesting can be arbitrarely deep. While I will not formally prove this, I will give a logicbased argument for its correctness:

  > Each `Stat` is compiled with an argument that contains a label with the location of the catch code that matches it. This label is initialised for each function as "unhandled", as soon as a `try` block is encountered the Stats in it, is compiled with a new label generated right there. This way the new label always propegate to the new code. Since this is determined at compile time it will not consume stackspace, so it is only limited by the memory of the compiler or any limitations in mosml, that I am yet unaware of. Note that this method of passing the catchlabel down is also what prevents crossfunction-catching from working, since each function is compiled independant from the others.
- Program #3:
  This program demonstrates the compilers inability to create crossfunction-catches. This essentially show the same bug as *except-find/quens*.

8 Strategy for improvement

Before I hit the deadline I was working on making the compiler able to do crossfunction-catches. I was working on a system where the catchlabel would be put on the stack using the `LA` and `SW` operations. It would then be retrived at the beginning of the function. My current problem is that I have yet to determine how to keep the adress alive since it is not in the registryallocator. I was considering modifying the registry-allocator to save a specific register to always save the adress of the last catch label. However I abandoned the idea because taking away a register just for that was impractical.

9 Conclusion

The correctness of the current implementation is argued for in a previus section, and based in the fact that all the testprograms behave as expected (some of them are expected to behave wrong since the compiler is not complete.) The correctness does not include the ability to perform crossfunction-catches. This proved to be the

most problematic part since where to jump cannot be determined at compiletime, but has to be done dynamically at runtime. A stack solution would be my best guess at how to implement this functino.

A Attached files

Attached to the assignment-reply is a zip file with the files needed to build the compiler, testprograms and the test.sh bash script.