Programming Paradigms Final Project: Building a Compiler in Haskell for the Sprockell

Group 33
Martijn Verkleij & Wouter Bos

University of Twente
m.f.verkleij@student.utwente.nl, w.f.a.bos@student.utwente.nl
s1466895 s1606824

July 7, 2017

Contents

Chapter 1

Introduction

In the Programming Paradigms course, the final project is a combination of compiler construction, functional programming, and concurrent programming. It requires its participants to build a compiler for a self-defined language, which targets SprIL, an in Haskell defined synthesized processor. The language has to support basic primitives like arithmetic expressions, 3GL control flow logic like if(-else) and while and must perform type checking on all expressions. Finally, basic concurrency must be incorporated in the language.

The language we (group 33) improved upon for this project is called Simple Haskell Language (SHL), with file extension *.shl*. We are both repeat students, and decided upon improving the best of our two languages. The final project (SHL2.0) supports all the required features, as well as some additional features. These additional features include: procedures, call-by-reference and enumerations. The entire compiler runs on Haskell, both the code-generation and front end. The Sprockell was slightly extended as well, for feature parity with last year's implementation and convenience.

Chapter 2

Summary

This chapter will give a summary of the features of SHL2.0.

Data types SHL supports three basic data types: integers, booleans and enumerations.

Simple expressions and variables SHL supports denotations for primitive values of types as well as operations for (in)equality for values of types. SHL is strongly typed and all variables are initialised upon declaration. It also supports scoping with variable shadowing. Operator precedence is adhered to. If operators of equal precedence occur, the expression is interpreted left-to-right.

The following expressions are supported:

- Parentheses
- Assignment
- Binary operation (with ==, !=, <>, &&, ||, <=, >=, <, >, +, -, *)
- Prefix Unary operation (with !, -, +)
- Variable
- Integer value
- Boolean value

Basic statements SHL supports the following statements:

- Block
- Declaration
- If
- While
- Call
- Fork
- Join
- Print
- Expression

Concurrency SHL supports global variables, fork and join statements to implement concurrency.

Procedures SHL supports basic procedures with call-by-reference. Procedures have no return value.

Chapter 3

Problems & Solutions

This chapter is a short discussion of some of the problems encountered during the project and their solutions.

3.1 Expressions

During the process of improving the compiler's expression handling, a couple of problems arose. Firstly, more types had to be declared in order to implement operator precedence in the tokenizer, grammar and AST-builder. These types included operators and grammar rules. New AST-node types were considered, but not needed. Furthermore, the new types had to be unambiguous, which meant defining other types than the ones used by the Sprockell and Haskell.
Secondly, defining the right grammar rules proved to be tricky. Since the compiler supports more operators than only plus, minus and asterisk, a classic LL(1) expression grammar had to be complemented with precedence rules for the other operators.
Thirdly, there was a small problem with the pattern matching when building the AST out of the ParseTree. It took some time to figure out the right typing in the pattern matches and the building of operator nodes had to be though out thoroughly.
Lastly, after modifying the ASTBuilder there was still a problem with consecutively recurring operator classes being right-associative instead of left-associative. A solution has been found in implementing a rebalancing function just before the checker phase, in which a left-rotation is performed on an AST if such consecutively recurring operator classes are present.

3.2 Enumerations

The second improvement and therefore the second challenge which gave some problems, was implementing Enumerations into the SHL compiler.
Firstly, Enumerations had to be recognised and translated into the AST. This proved to be easier than expected. The real challenge was in type-checking and code-generation. An extra EnumerCheckerType has been added to be able to perform type-checking. This type-checking has then been implemented in the checker. Because of the extra checker-type, the code had to be changed in a lot of places, which was more time-consuming than planned.
Secondly and lastly, the generation of code was implemented. One problem in particular proved to be tricky in this stage, and that was the fact that initially the declared enumeration values

were not unique and therefore, the first value of two different enumeration were equal in the perspective of the Sprockell. This has been fixed by giving each enumeration value a unique id.

3.3 New Sprockell

Luckily we had few problems with the new version of the Sprockell. The most important change was the generated code to output strings out of the Sprockell. After changing one instruction name and after checking that every character was outputted correctly, this problem also has been solved.

3.4 Time

A big factor in each and every project is time. For this project a very specific time schedule has not been made. A generic planning including a task-list and task-division has been discussed multiple times through the process of improving SHL. Eventually, the amount of improvements are less than hoped for because more time has been spent on the re-take exam for the Concurrent Programming strand of the Programming Paradigms module.

Chapter 4

Sprockell Extensions

All extensions which are mentioned below, are now implemented in Sprockell and are extensions from last year Sprockell's version. No alterations to the code generation were needed to correspond with the extensions discussed in this chapter.

4.1 ComputeI Instruction

A compute with immediate value, `ComputeI operator reg value reg`. This instruction takes a register and a value to do a computation with the operator, which is then written to the second register. It is mostly used to calculate ARP offsets. This instruction was inspired by the *iloc* instructions using an immediate value (eg. `AddI, SubI`.

4.2 Incr4 & Decr4

Two `Operators` were added to the alu operators: Incr4 and Decr4. They were originally added to more easily do the steps of four to calculate offsets, before `ComputeI` was added. They are currently unused.

Chapter 5

Detailed Language Description

This chapter will describe every feature of SHL in detail: providing a basic description; information on the syntax with at least one example; usage information along with restrictions; a description of its effects and execution; and some general information on the generated code.

5.0.1 Program

Syntax

`[GLOBAL]...[ENUM]...[PROCEDURE]...[STATEMENTS]...`

GLOBAL Global variable declarations as defined in 5.0.2 (Global)
ENUM Enumeration declarations as defined in 5.0.3 (Enumeration)
PROCEDURE Procedures as defined in 5.0.4 (Procedure)
STATEMENTS Statements as defined in 5.1 (Statements)

Example

```
global int number = 5;

enum foo = {bar,baz};

procedure eq(int num1, int num2, bool out) {
    if (num1 == num2) {
        out = true;
    } else if (bar == bar) {
        out = false;
    }
}

int otherNumber = 6;
bool out;
eq(number, otherNumber, out);
print(out);
```

Usage

All files must follow the program syntax, and may only contain a single program.

Semantics

A program is a collection of code which can be used to create an executable set of instructions. It is the root node of the Abstract Syntax Tree.

Code Generation

Any program is generally built up as follows:

- **Thread control code**
  - **Thread Control Loop**
    All extra threads loop here, waiting to accept fork calls.
  - **PreCall forked procedures**
    Once a sprockell accepts a fork call, it reads the AR from global memory, and starts execution.
  - **PostCall forked procedures**
    Once an auxilary sprockell finishes the procedure, this code handles cleanup.
- **Procedures 5.0.4**
- **Global declarations 5.0.2**
- **Main code**
- **Post-program code**
  Stops auxilary threads.

5.0.2 Global

Syntax

```
global <TYPE> <ID> [= <EXPRESSION>] ;
```

TYPE A type as defined in 5.3.1 (Types)
ID A string as defined by 5.2.2 (Variable)
EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
global bool flag = true;
global int number;
```

Usage

Used to declare global variables and an optional assignment. The type of the expression must match the type of the global variable.

All global variables MUST be defined at the top of the program, even before procedures.

The id of a global variable is unique in the whole program. No other variable of procedure may use the same id. They can therefore not be shadowed.

Semantics

The global variable declaration reserves a space in shared memory and writes a value to it. All global variables are initialized to the default value (see 5.2.3 (Integer) and 5.2.4 (Boolean)) if no value is explicitly assigned.

Global variables are reachable from anywhere below its declaration in the code, in any thread. Beware, however, that the variable can be written to from any thread as well, and using the shared memory is significantly slower than using the local memory.

Globals that are passed as arguments to a procedure have their own intricacies, see 5.3.3 for more information.

Code Generation

All globals are saved in global memory. Writes and reads to globals are done atomically, but assignments are not. This avoids data races, but to ensure atomicity on the whole assignment, one must implement his own mutual exclusion.

5.0.3 Enumeration

Syntax

```
enum <ID> = { <VAR> [, <TYPE> <VAR>]...)  }
```

ID A string as defined in 5.2.2 (Variable)
VAR A string as defined in 5.2.2 (Variable)

Examples

```
enum foo = {bar}
enum test = {test1, test2}
```

Usage

Used to declare an enumeration. Enumerations are all equal, as in, the compiler does not type check individual enumerations from each other, due to time constraints.

The ID of an enumeration, a well as it's values must be unique in the program.

Semantics

Enumerations are usable everywhere in the program. Enumerations can only be checked for equality with each other.

Code Generation

During code generation, all enumerated values are mapped onto integer values. Literal enum values emit this mapped integer value. As they are type checked for comparison by equality only, arithmetic operations are impossible. For their declaration, no code is emitted.

5.0.4 Procedure

Syntax

```
procedure <ID> ( [<TYPE> <VAR>] [, <TYPE> <VAR>]...)  <STATEMENT>...
```

ID A string as defined in 5.2.2 (Variable)
TYPE A type as defined in 5.3.1 (Types)
VAR A variable as defined in 5.2.2 (Variable)
STATEMENT A statement as defined in 5.1 (Statements)

Examples

```
procedure empty() print(0);
procedure other(int num, bool flip) {
    while ((num > 0)) {
        num = --num;
        flip = !flip;
    }
    print(num, flip);
}
```

Usage

Used to declare a procedure. Because call-by-reference (see 5.3.3 (Call-by-reference)) is used, a variable passed as an argument can be used to write resulting values. One could also use the global variables (see 5.0.2 (Global)), as they are accessible from everywhere.

The id of a procedure is unique in the whole program. No other variable of procedure may use the same id.

Semantics

A procedure is section of code that can be executed from anywhere, using a call or fork statement (see 5.1.6 (Call) and 5.1.4 (Fork)) and passing the appropriate number of arguments to it.

Code Generation

Procedure code has the following structure:

- **PostCall code**
  Copies all arguments into the local data area for use.
- **Procedure's statements**
- **PreReturn code**
  The final result of all arguments is read, and if they are global or local variables, saved to the appropriate location.

5.1 Statements

5.1.1 Declaration

Syntax

```
<TYPE> <ID> [= <EXPRESSION>] ;
```

TYPE A type as defined in 5.3.1 (Types)
ID A string as defined in 5.2.2 (Variable)
EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
int number = 1+1;
bool flag;
```

Usage

Used to declare local variables and an optional assignment. The type of the expression must match the type of the variable.

The id of a variable is unique in the scope it is defined in. No other variable in that scope may use the same id.

Semantics

The variable declaration writes the value of the variable to the Local Data Area of where it was (most recently) defined. All variables are initialized to the default value (see 5.2.3 (Integer) and 5.2.4 (Boolean)) if no value is explicitly assigned.

Code Generation

A declaration evaluates the expression behind it before assigning it to the variable. The variable's value is saved in the appropriate Local Data Area, by following the scopes upward until the right scope is reached, after which it is saved with an offset.

5.1.2 If

Syntax

```
if ( <EXPRESSION> ) <STATEMENT> [else <STATEMENT>]
```

EXPRESSION An expression as defined in 5.2 (Expressions)
STATEMENT A statement as defined in 5.1 (Statements)

Examples

```
if (flag) {
    // do something
}
if (flag) print(flag); else {
    // do something
}
```

Usage

Execute a section of code based on an expression. The type of this expression must be a boolean.

Semantics

If the expression evaluates to *true*, execute the first statement. If it evaluates to *false*, execute the code after the first statement, which can be either the second statement or the code that comes after the if statement.

Code Generation

First the expression will be evaluated, after which a branch jumps to the else block upon a false result, or to the next expression if no else block is present. After the if-block, a jump to after the else block is placed when an else block is present.

5.1.3 While

Syntax

```
while ( <EXPRESSION> ) <STATEMENT>
```

EXPRESSION An expression as defined in 5.2 (Expressions)
STATEMENT A statement as defined in 5.1 (Statements)

Examples

```
while (flag) {
    // do something
}
```

Usage

Execute a section of code while the expression is *true*. The type of this expression must be a boolean.

Semantics

If the expression evaluates to *true*, execute the statement. Repeat this for as long as the expression keeps evaluating to *true*.

Code Generation

As long as the expression is true, the code in the following block will be executed. The expression is re-evaluated after each execution of the block.

5.1.4 Fork

Syntax

```
fork <ID> ( [<EXPRESSION> [, <EXPRESSION>]...]  )  ;
```

ID A string as defined by 5.2.2 (Variable)
EXPRESSION An expression as defined by 5.2 (Expressions)

Examples

```
fork proc0();
fork proc1(flag);
fork proc2(5, flag = true);
```

Usage

Run a procedure, which must have been declared somewhere, on a separate thread. The expression types must match the types defined during the procedure declaration (see 5.0.4 (Procedure)).

Semantics

Writes the argument to shared memory and tells the thread pool to start parallel execution of the procedure.

Beware, if more procedures are given to the thread pool than there are threads, fork may have to wait for a thread to finish its work before continuing execution. In the case where programs with forks are executed on only one sprockell, it will deadlock.

Code Generation

A fork call must start a procedure in another thread. To accomplish this, the first 30 addresses in global memory are reserved for this purpose. A further few addresses, namely as much as there are threads, are reserved for an occupation bit. In practise this means that procedures with more than 7 arguments cannot be forked without memory corrupting.

The addresses are reserved as follows:

- **End flag**
  Set when the auxilary threads may cease execution. Each auxilary sprockell checks this record before attempting to read an AR from shared memory
- **Wr flag**
  Used to set the AR space as occupied by an AR that may be executed. An auxilary threads sets it to 0 when it has read the AR.
- **Rd flag**
  Read-protects the AR as it is being written, and to exclude other sprockells from handling the request.
- **Jump index**
  line number of the procedure to execute.
- **Argument count**
  Number of arguments that follow.
- **Arguments...**
  - **Value**
  - **Local address**
    The address to write the result back to, if the argument has one. For example, an expression has none.
  - **Global address**
    Idem.

5.1.5 Join

Syntax

```
join ;
```

Example

```
join;
```

Usage

Ensures all auxilary threads have finished execution before continuing.

Semantics

Blocks execution of the main thread until all other threads have finished their work. May only be called in code that is not executed in auxilary threads, for example by calling a procedure with fork that contains a join statement.

Code Generation

A join statement iterates over the occupation bits of all auxilary threads. Only if they are all zero, this statement may end. It will therefore throw an error when called from an auxilary thread, as can happen when used in a procedure.

5.1.6 Call

Syntax

```
<ID> ( [<EXPRESSION> [, <EXPRESSION>]...]  )  ;
```

ID A string as defined by 5.2.2 (Variable)
EXPRESSION An expression as defined by 5.2 (Expressions)

Examples

```
proc0();
proc1(flag);
proc2(5, flag = true);
```

Usage

Execute the called procedure sequentially, which must have been declared. The expressions must have the same types as the procedure's as defined in its declaration (see 5.0.4 (Procedure))

Semantics

Go to the procedure code and execute the procedure with the expressions, then return to the call.

Code Generation

Calls a procedure sequentially. Any separate variables that are given as an argument are handled call-by-reference. Sets up the AR before jumping to the procedure.

5.1.7 Expression

Syntax

```
<EXPRESSION> ;
```

EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
a = (5 + (--b));
true;
-+a;
++a;
```

Usage

Allows expressions to be executed as statements. The only useful purpose is to increment or decrement a variable with a prefix unary operator.

Semantics

Execute the expression, this generally has no effects, except for an increment or a decrement of a variable.

Code Generation

As expressions have a result on stack, one must pop this value if the expression is defined as a statement. Otherwise the stack would fill up. One special case has been made for increments and decrements.

5.1.8 Block

Syntax

```
{ [STATEMENT]...}
```

STATEMENT A statement as defined in 5.1 (Statements)

Example

```
{
    int i = 0;
    {
        i = ++i;
        {
            int i = 5;
        }
        print(i == 1);
    }
    print(i == 1);
}
```

Usage

A block is a single statement that contains zero or more statements. It is mostly used within procedures and statements to executes more than one statement.

Semantics

A block opens a new scope, then executes the code within. When exiting a block, the scope is closed.

Code Generation

As any block opens a new scope, a new mini-AR is created for each scope. It only contains a pointer to it's parent's AR and all variables declared in this scope in a Local Data Area.

5.1.9 Print

Syntax

```
print ( <EXPRESSION> [, <EXPRESSION>]...)  ;
```

EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
print(a);
print(true, 5, 1983);
print(a = ++a, (11 - 2) * a));
```

Usage

Prints values of evaluated expressions to the console.

Semantics

Evaluates the expressions and prints the values as they appear in memory, meaning a boolean is represented as either a zero (*false*) or a one (*true*).

Code Generation

As SprIL does not have a print instruction, a new one was created and added to the Sprockell source code. See also chapter 4. This statement simply evaluates all expressions in its arguments and prints them one per line.

5.1.10 Assignment

Syntax

```
<ID> = <EXPRESSION>
```

ID A string as defined in 5.2.2 (Variable)
EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
a = 5;
b = c <> d && e;
```

Usage

Used to assign a value, in the form of an expression, to a variable. The variable must have been declared beforehand, and may be either global or local. The type of the expression must match the type of the variable.

Semantics

Assignment evaluates the expression and writes it to the address of the variable.

Code Generation

First, the expression will be evaluated. The result is assigned to the variable, may it be in local or global memory. These are resolved by AR traversal and lookup in a static table respectively. The result is pushed to the stack.

5.2 Expressions

5.2.1 Parentheses

Syntax

```
( <EXPRESSION> )
```

EXPRESSION An expression as defined in 5.2 (Expressions)

Example

```
a = -(-(--a); // the same as: a = (-1) * (-1) * (a - 1);
```

Usage

Parentheses are used to enforce which operator is used (see the example above). It can also be used to enforce the order in which an expression is evaluated, but since this already explicitly happens (see 5.2.6 (Operation)) it should not be necessary to use a parentheses expression for it.

Semantics

Everything between the parentheses is evaluated and the value is returned as the result of this expression.

Code Generation

Required around any binary expression and optional around unary expressions to, as in the example, differentiate between the negation and decrement operator. The result of the inner expression is pushed to stack.

5.2.2 Variable

Syntax

`<ID>`

ID A string, starting with a letter, which may use any alphanumerical character in addition to the following characters: ~'"@#$\.?:_

Examples

```
a
a@__b"42"\#1337'
```

Usage

A variable must be declared (see 5.1.1 (Declaration)) before use. It has a type which is determined upon declaration.

Semantics

Evaluation of a variable returns its value.

Code Generation

The variable's value is looked up in the AR stack or loaded from global memory and pushed to stack.

5.2.3 Integer

Syntax

`<INTEGER>`

INTEGER An integer string

Examples

```
42
1337
0000004201337
```

Usage

Takes the value of the integer, removes leading zeros.

Semantics

Upon evaluation it returns its integer value.

Code Generation

Its value is pushed to stack.

5.2.4 Boolean

Syntax

`<BOOLEAN>`
BOOLEAN Where a boolean is either "`true`" or "`false`"

Examples

```
true
false
```

Usage

Takes the value of the boolean (either one or zero) and returns it.

Semantics

Upon evaluation, return the corresponding binary representation of the boolean, where *false* equals zero and *true* equals one.

Code Generation

Its value is pushed to stack.

5.2.5 Enumeration

Syntax

`<ENUM>`
ENUM A string that is equal to a declared enum value.

Examples

```
foo
```

Usage

Pushes the mapped enum value to the stack.

Semantics

Upon evaluation, return the corresponding mapped value of the enum value.

Code Generation

Its value is looked up in a generated enum mapping table, and pushes its value to stack.

5.2.6 Operation

Syntax

`<EXPRESSION> <OPERATOR> <EXPRESSION>`

EXPRESSION An expression as defined in 5.2 (Expressions)

OPERATOR One of the following operators: `==`, `!=`, `&&`, `||`, `<>`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*` (see 5.3.2 (Operators))

Examples

```
(true <> b)
((a + b) == (c + d))
```

Usage

Apply operator on two expressions. Both expressions must be of the same type, which must also match one of the types supported by the operator.

Semantics

After both expressions have been evaluated, the operation is evaluated and its result will be returned.

Code Generation

It's left- and right hand side are evaluated, and its results are used as arguments to the operation. The result of the operation is pushed to stack.

5.2.7 Prefix Unary Operation

Syntax

`<OPERATOR> <EXPRESSION>`

OPERATOR One of the following operators: `--`, `++`, `+`, `-`, `!` (see 5.3.2 (Operators))

Examples

```
!b
-(--a)
---a // is the same as: --(-a)
```

Usage

Apply operator on the expression. The expression type must match one of the types supported by the operator.

Semantics

After the expression has been evaluated, the operation is evaluated and its result will be returned.

Code Generation

It's right hand side is evaluated, fed to the operator and its result is pushed to stack.

5.3 Other Features

5.3.1 Types

Syntax

`<TYPE>`
TYPE `int`, `bool` or `enum`

5.3.2 Operators

Syntax

`<OPERATOR>`
OPERATOR One of the following: `==`, `!=`, `&&`, `||`, `<>`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*`, `-`, `++`, `!`

Usage

OPERATOR Operation: `supported types` → `return type`
== equals: `int, bool` → `bool, enum` → `bool`
!= not equals: `int, bool` → `bool, enum` → `bool`
&& and: `bool` → `bool`
|| or: `bool` → `bool`
<> xor: `bool` → `bool`
<= lesser than or equals: `int` → `bool`
>= greater than or equals: `int` → `bool`
< lesser than: `int` → `bool`
> greater than: `int` → `bool`
+ add: `int` → `int`
- subtract: `int` → `int`
***** multiply: `int` → `int`
– decrement: `int` → `int`
++ increment: `int` → `int`
! not: `bool` → `bool`

5.3.3 Call-by-reference

SHL uses call-by-reference on calls to procedures. This is almost essential to make procedures useful, as the only other option to communicate values between a procedure and its caller would be through global memory.

It is implemented by passing an optional return address in shared and local memory to write the result back to after the procedure is complete. In theory an argument could be written to both at the same time, as a memory slot is used for each (and makes them distinguishable). These are used to write the results back to their memory locations before returning to the caller.

To make use of it, pass any variable as a naked argument to a call to any procedure in the program. Expressions like `(i+3)` or `i = 12` do NOT work, as the argument will only be seen as an expression and will therefore not have a memory location associated with it. Keep in mind that any global variables passed as an argument will be written to global memory upon completion of the procedure. This does not influence any assignment inside the procedure, these are still done as they are evaluated.

5.3.4 Error Handling

The SHL compiler does not support proper exception handling, but does throw errors of varying usefulness. During the tokenization phase, the only error thrown is an illegal character error.

During the parsing phase, the only error which might be thrown is a token list not fully parsed error, indicating the grammar cannot parse the token list.

The checker phase thrown different kinds of errors, all related to context constraints, they generally indicate the function which throws the error as well as printing some of the responsible data.

The code generation and runtime phases thrown the following kinds of errors:

Upon code generation, an error is thrown when a user attempts to compile code with a fork statement with the intention of using only one thread.

During runtime, when a join statement is executed by a thread other than the main thread, it prints an error code and ceases operation. This may turn out unhelpful, as the other threads are not notified and useful operation ceases. Since the occupation bit cannot be unset at this stage, any subsequent join statement will deadlock. Any values it would have returned are lost.

Chapter 6

Description of the Software

The compiler consists of a number of Haskell files, and some additional files. This chapter will go over the functions of each of those files. It is assumed that the Sprockell has been installed from the GitHub repository which has been referenced by the teachers.

6.1 ASTBuilder.hs

The purpose of the ASTBuilder is to build an Abstract Syntax Tree using a parsetree. The ASTBuilder also contains the functions to convert an AST to a RoseTree with or without debug information.

6.2 Checker.hs

The checker does type checking on an AST and adds information about scopes (symboltable) to it. It works in two passes, first collecting information about global variables and procedures, then checking for all context constraints.

6.3 CodeGen.hs

CodeGen takes a checked AST and generates a program of SprIL instructions, runnable on Sprockells.

6.4 Constants.hs

Constants stores constant values used in the code generation. Simply aliases for memory addresses and offsets.

6.5 FP_ParserGen.hs

Parser generator supplied by the course.

6.6 Grammar.hs

Grammar contains the grammar used in the compiler.

6.7 Main.hs

Main file, used for compilation and execution of SHL programs. Read the README.md for information on how to use it.

6.8 README.md

Contains some information about the project in general (eg. the Trello board) and instructions on how to use the compiler.

6.9 Test.hs

Used for internal testing, contains functions to print and write debug information, show ASTs with and without debug information, show the parse tree, and show the token list.

6.10 Tokenizer.hs

Tokenizer tokenises a string into a list of tokens.

6.11 Types.hs

Contains all the Haskell types used during compilation, including Alphabet, AST, and checking/scope types.

Chapter 7

Test Plan & Results

7.1 Implemented Tests

Following is a list of all the test files that have been used to test the compiler, and a short description of their purpose.

syntax1 Tests incorrect program syntax
syntax2 Tests incorrect procedure syntax
syntax3 Tests incorrect variable syntax
syntax4 Tests incorrect if syntax
syntax5 Tests incorrect expression syntax
wrong_type Tests whether a wrong type is detected
not_declared Tests whether a variable which has not been declared is detected
cyclic_recursion Tests for correct cyclic recursion
deep_expression Tests for correct evaluation of nested expressions
fib Tests for correct evaluation of a Fibonacci procedure
if Tests a correct simple if statement
ifelse Tests a correct simple if-else statement with some additional scoping
infinite_busy_loop Tests behaviour in an empty infinite loop
infinite_loop Tests behaviour in an infinite loop with some operation in it. Also tests integer overflows, which are not detected.
nested_procedures Tests for correct evaluation of nested procedures
recursion Tests for correct recursion
while Tests a simple correct while statement
enum Tests simple enumerations
call_by_reference Tests for correct multi-threaded call-by-reference
blocks Tests for correct handling of scopes
simple_proc Tests a simple correct procedure
banking Tests a concurrent banking application
peterson Tests for correct evaluation of Peterson's algorithm
simple_concurrency Tests a simple correct concurrent program
multiple_globals Tests behaviour of concurrent printing of global variables
join_test Tests whether join behaviour is correct

The source code and results of all tests have been documented in `testreport.pdf`.

7.2 Test Plan

The testing has been roughly divided into three cases: syntax, context constraints and semantics. For the first two phases most of the testing of correct code occurs during the semantic testing and as informal testing during the building of those parts of the compiler. Some additional tests have been written to more formally test the incorrect code.

The shape of the parse tree and Abstract Syntax Tree have been extensively observed and checked during the building of the checking part of the compiler. This has mostly been done by slightly tweaking a program a multitude of times, to produce all intended shapes of the tree and attempting to produce unintended shapes, and building the trees. This part of the testing, as well as the previous part, have not been documented very well, and might therefore appear somewhat lacking compared to the semantic testing.

The semantics, or run-time, testing has been given the most time and effort, and checks for correctness of code generation and intended behaviour. Since very little run-time error are thrown (see 5.3.4 (Error Handling)), there are only a few tests of incorrect code, or code producing unintended effects.

7.3 How To Run a Test

To run a test, simply follow the `README.md`, using the following path: `test/<fileName>`, where `fileName` is one of the tests described above. Remember that for a concurrent program, which is any program that uses at least one fork statement, multiple Sprockells have to be used.

Chapter 8

Personal Evaluation

8.1 Martijn

The project is actually quite fun to do once you have a better understanding of the time needed to deliver a working product. It helps too that stress factors like an FP-resit or falling behind on exercises are just not there. Even though I had to take the CP resit, most other grades were excellent and help in achieving a good average grade. The module as a whole is a lot of work, but is a lot more doable as a second-timer. Grades are better, workload is better.

I am a great fan of the parser generator functional programming exercises, they were a change in pace for us, but gave us the confidence to do this project fully in Haskell which turned out to be a great choice in terms of motivation and having the overview. With ANTLR I feel like talking to a black box (even though ANTLR is easier to define a language in due to left-recursiveness, LL_X and so on). The new Logic programming exercises and project are a great addition too, as they show more of the power of logic programming than last year's did.

That's all. As far as I can see we did our best. Let's hope it is enough.

8.2 Wouter

In my opinion, this project is a really good and fun way to test if we have gained the knowledge needed to pass the module. It covers almost all aspects: concurrency, functional programming and compiler construction. For me it is really fun to actually do something with all concepts I've learned.

Doing the module and the project for the second time gave me an even better insight in all concepts relevant with the three main topics of this project. Especially my opinion on functional programming has changed, despite my mathematical skills being not that great.

Unfortunately, I've not been able to put the time in the project that I wanted to put in. This is mainly because of the CP-retake and the second homework assignment of CC. In my opinion, this module should be combined with the AI strand of module 6 and given for one full semester instead of only 1 quarter.

What went well in the project is tokenizing, parsing and AST-building of the expressions. I definitely consider programming the first and second phase of compilers as one my stronger skills. Something I could work on is improving the programming of the code-generation phase.

All in all, I have had a great time doing this module, despite of repeating it.

Appendices

Appendix A

Grammar Specification

```
grammar nt = case nt of

    -- Program
    Program ->  [[ (*:) [Global], (*:) [Enum], (*:) [Proc], (*:) [Stat] ]]

    -- Globals
    Global  ->  [[ global, Type, Var, (?:) [ass, Expr], eol ]]

    -- Enumerations
    Enum    ->  [[ Type, Var, ass, lBrace, Var, (*:) [comma, Var], rBrace, eol ]]

    -- Procedures
    Proc    ->  [[ procedure, Pid, lPar, (?:) [Type, Var, (*:) [comma, Type, Var]], rPar, Stat ]]

    -- Statements
    Stat    ->  [[ Type, Var, (?:) [ass, Expr], eol ]                              -- declaration
                ,[ ifStr, lPar, Expr, rPar, Stat, (?:) [elseStr, Stat] ]           -- if
                ,[ while, lPar, Expr, rPar, Stat ]                                 -- while
                ,[ fork, Pid, lPar, (?:) [Expr, (*:) [comma, Expr]], rPar, eol ]   -- fork
                ,[ join, eol ]                                                     -- join
                ,[ Pid, lPar, (?:) [Expr, (*:) [comma, Expr]], rPar, eol ]         -- call
                ,[ Expr, eol ]                                                     -- expression
                ,[ lBrace, (*:) [Stat], rBrace ]                                   -- block
                ,[ printStr, lPar, Expr, (*:) [comma, Expr], rPar, eol ]           -- print
                ,[ Var, ass, Expr, eol ]]                                          -- assign

    -- Expressions. Already implements the fact that some operations are bound tighter than
    -- other operations (operator precedence). The order from tightest to loosest: Mul, PlusMin,
    -- Ord, Equal, And, XOR, Or.
    Expr      ->  [[ OR, (?:) [Expr'] ]]                -- Expr -> OR (?:) Expr'

    Expr'     ->  [[ opOr, OR, (?:) [Expr'] ]]          -- opOr, OR (?:) Expr'

    OR        ->  [[ XOr, (?:) [OR'] ]]                 -- XOr, (?:) Or'
```

```
OR'       ->  [[ opXor, XOr, (?:) [OR'] ]]          -- opXor, XOr, (?:) OR'

XOr       ->  [[ AND, (?:) [XOr'] ]]                -- AND, (?:) XOr'

XOr'      ->  [[ opAnd, AND, (?:) [XOr'] ]]         -- opAnd, AND, (?:) XOr'

AND       ->  [[ EQUAL, (?:) [AND'] ]]              -- EQUAL, (?:) AND'

AND'      ->  [[ opEqual, EQUAL, (?:) [AND'] ]]     -- opEqual, EQUAL, (?:) AND'

EQUAL     ->  [[ Ord, (?:) [EQUAL'] ]]              -- Ord, (?:) EQUAL'

EQUAL'    ->  [[ opOrd, Ord, (?:) [EQUAL'] ]]       -- opOrd, Ord, (?:) EQUAL'

Ord       ->  [[ Term, (?:) [Ord'] ]]               -- Term, (?:) Ord'

Ord'      ->  [[ opPlusMin, Term, (?:) [Ord'] ]]    -- opPlusMin, Term, (?:) Ord'

Term      ->  [[ Factor, (?:) [Term'] ]]            -- Factor (?:) Term'

Term'     ->  [[ opMul, Factor, (?:) [Term'] ]]     -- OpMulDiv Factor (?:) Term'

Factor    ->  [[ lPar, Expr, rPar]                  -- (Expr)
              ,[ PreUnary, Expr ]                   -- Prefix
              ,[ Var ]                              -- Var
              ,[ IntType ]                          -- Int
              ,[ BoolType ]]                        -- Bool

-- Other
PreUnary  ->  [[ opPlusMin ]                        -- Prefix unary + and -
              ,[ opIncDec ]                         -- Prefix unary ++ and --
              ,[ opNot ]]                           -- Prefix unary !
```

```
Type      ->  [[ typeStr ]]                    -- type
Var       ->  [[ var ]]                        -- variable
Pid       ->  [[ Var ]]                        -- procedure identifier
IntType   ->  [[ intType ]]                    -- number
BoolType  ->  [[ boolType ]]                   -- boolean
```

Appendix B

Extended Test Program

The extended test program shown here is Peterson's algorithm. It shows how, using the available methods for concurrency, two thread using the same variable have mutually exclusive access to it.

B.1 Peterson's Algorithm Test

```
1  global bool flag_0 = false;
2  global bool flag_1 = false;
3  global int turn = 0;
4  global int i = 0;
5
6  enum test = {die};
7
8  procedure p_0() {
9      flag_0 = true;
10     turn = 1;
11     while ((flag_1 && (turn == 1))) {
12         // wait
13     }
14     // begin critical section
15     int j = 5;
16     while ((j > 0)) {
17         i = ++i;
18         j = --j;
19     }
20     // end critical section
21     flag_0 = false;
22 }
23
24 procedure p_1() {
25     flag_1 = true;
26     turn = 0;
27     while ((flag_0 && (turn == 0))) {
28         // wait
```

```
29      }
30      // begin critical section
31      int j = 5;
32      while ((j > 0)) {
33          i = --i;
34          j = --j;
35      }
36      // end critical section
37      flag_1 = false;
38  }
39
40  procedure test1(int j) {
41      while ((j > 0)) {
42          fork p_0();
43          fork p_1();
44          join;
45          print(i);
46
47          fork p_1();
48          fork p_0();
49          join;
50          print(i);
51
52          j = --j;
53      }
54  }
55
56  test1(10);
```

B.2 Correct Executions

Every time a value is printed in Peterson's algorithm test, it should be zero. The output of a simulation on the Sprockell with our compiler:

```
What file do you want to run? Please provide the relative path excluding the extension.
test/peterson
How many Sprockells do you want to use to run this file?
3
Running: test/peterson.shl
On 3 Sprockells
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
```

```
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
```