CS644 Project Report: A1

Submitted in partial fulfillment of the requirements for the course

Submitted by

Name	ID
Siwei Yang	s8yang
Justin Vanderheide	jtvander
Jian Li	j493li

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada – N2L 3G1

Winter 2015

1 Implementation

1.1 Scanner

The scanner is implemented in the file *Scanner.hs*.

Each token type has an associated function that implements a recognizer for it. For example *scanChar* recognizes char literals, and *scanBool* recognizes boolean literals. Each of these functions consumes a string containing the unscanned portion of the file, and returns a Maybe representing the result.

If the recognizer did not find its token type, it returns Nothing. For example if the input is "while (..." then *scanDecimalInteger* will return Nothing since the beginning of the input is not a valid integer literal token. If the recognizer does find its token type then it returns a pair where the first element is the token type, and the second element is the lexeme of the token. For example in the input "while (..." the *scanKeyword* function will return ("KEYWORD", "while"). Each of these functions is run in parallel, and the longest lexeme is taken.

```
*Scanner> scanDecimalInteger "while ("
Nothing
*Scanner> scanKeyword "while ("
Just ("KEYWORD","while")
```

Sometimes there are ties for the longest lexeme, for example both the identifier and keyword scanners will return the lexeme "while". Theoretically, the identifier scanner should reject all the reserved keywords. However, this adds unnecessary complexity to the program as a simple precedence rule suffice: ties are broken by the ordering of the scanners in the list of scanners we provide to the scannerRunner. By placing the keyword scanner before the identifier scanner, keywords are correctly tokenized instead of being mistreated as identifiers. After finding the desired token the lexeme is removed from the input string and the process repeats until the input string is empty. The result is a list of Tokens that we can feed into the parser.

1.2 Parser

Thanks to the provided tool, the generation of state transitions is taken care of. Most of our effort was, thus, put into the creation of a valid grammar. Besides following the guidelines of creating grammar of rightmost derivations, we also created our own tools to facilitate rapid developing and testing grammars. We will talk more about that in the challenges section.

Once we had our first grammar, we took the standard approach of implementing a LALR machine: stack DFA. Fortunately, Haskell is a high level language with support to algebraic type, which made implementing state machine a trivial task. We, firstly, create types to match DFA definitions.

```
type TransitionKey = (Int, String)
type TransitionVal = (Bool, Int)
type Transition = (TransitionKey, TransitionVal)
type TransitionTable = Map TransitionKey TransitionVal

data DFA = DFA {
```

```
    states :: [Int],
    units :: [AST],
    numStates :: Int,
    rules :: TransitionTable,
    productions ::[Production]
}
```

Then, each transition of the DFA is a matter of finding the right transition from TransitionTable, and act on the DFA accordingly. Since we only care about the end result of the parse, we wrote the execution of the DFA in CSP style for its simplicity.

```
run :: (DFA, [AST]) -> (DFA, [AST])
run (dfa, []) = (dfa, [])
run (dfa, (ast:rst)) = if isNothing lku then (dfa, (ast:rst))
                       else if changeState
                       then run ((DFA (num:ss) (ast:u) ms rules prods), rst)
                       else run ((DFA nss nu ms rules prods), (nast:ast:rst))
    where
        DFA ss u ms rules prods = dfa
        tk = name ast
        s = head ss
        lku = lookup (s, tk) rules
        Just (changeState, num) = lku

        prod = prods !! num
        units = snd prod
        nast = AST (fst prod) (take (length units) u)
        nss = drop (length units) ss
        nu = drop (length units) u
```

Note that the DFA will continuously accept AST tokens from the provided list. Therefore, by checking whether the list is exhausted after execution of the DFA, we are able to determine if the list of tokens can be successfully parsed. Given the parsing succeeded, we can retrieve the parse tree from DFA as it was built during the execution.

1.3 AST Construction

After getting the concrete syntax tree from parser, we simplify it to abstract syntax tree(AST) by removing all redundant structures and info only used at parsing stage. The AST construction functions are implemented in the file *AST.hs*. We define the following data types to represent different grammar structures for AST: *Statement*, *CompilationUnit*, *TypeDec*, *TypeDec*, *Field*, *Method*, *Constructor*, *StatementBlock*, *TypedVar*, *Expression*, *Name* and *Arguments*. For example, data type *Statement* is an abstract representation for statement in the grammar.

```
data Statement = LocalVar {localVar :: TypedVar, localValue :: Expression}
               | If { ifExpression :: Expression, ifBlock :: StatementBlock,
                  elseBlock :: Maybe StatementBlock}
               | While { whileExpression :: Expression, whileBlock ::
                  StatementBlock}
```

```
                | For { forInit :: Maybe Statement, forExpression :: Maybe
                    Expression, forStatement :: Maybe Statement, forBlock ::
                    StatementBlock}
                | Block StatementBlock
                | Expr Expression
                | Return (Maybe Expression)
                | Empty
```

For each input *AST*-type tree, we need to look at its name and productions to determine its structure. All elements in production are still *AST*-type trees, we just construct them recursively. Meanwhile, we can also get some types during expression construction, since we have type keywords and type of tokens.

```
data Type = TypeByte | TypeShort | TypeInt | TypeChar | TypeBoolean
          | TypeString | TypeNull | TypeVoid
          | Object Name
          | Array Type
```

For example, we can give a *TypeInt* if we meet a keyword "Int", and give a type *TypeBoolean* if we meet a token "True" with type "LITERAL_BOOL". This part can be extended to do type inference for semantic analysis.

1.4 Weeding

The weeder is implemented in the file *Weeder.hs*. There are 'weed' methods for each data type that makes up our AST, from the top-most 'CompilationUnit' to the bottom-most 'Expression'. Each weed method checks the current AST entry and then also calls weeders for the children. For example the 'weedMethod' weeder checks the return type, name, modifiers, and signature then calls weedStatements on the body. The weeders make extensive use of Haskell's pattern matching and guards which make it easy to detect weeds. The compilation runner passes the CompilationUnit into weedCompilationUnit which then traverses the whole AST. If an error is found a pair of (AST, String) is returned. The string explains the problem that was found, and the AST is the item that triggered the error so that location can be communicated.

2 Challenges

2.1 Scanning

Scanning was very straightforward, the only trouble we ran into was making sure that keywords were scanned as keywords, not identifiers, which we did by prioritizing keywords in the event of a tie.

2.2 Parsing

For starters, the '.cfg' input format for the LR1 generator wasn't very nice to work with. In order to make our lives easier we expressed our grammar in a more human readable format (see 'tools/joos1w.bnf'). The joos1w.bnf files is preprocessed by 'tools/simplify.rb' which outputs a .cfg file for the LR1 generator which in turn generates the .lr1 file for our compiler. This preprocessing takes care of the accounting stuff like what are the terminal and nonterminal symbol, and numbers of them. It also supports marking elements of the production rules as optional by encasing them in pipes: $|Optinal|$.

Coming up with a valid grammar was more involved. In particular, it was difficult to create an unambiguous grammar that was easy to understand. So we had to make some compromises. Some of the sacrifices we had to make in order for the grammar to be unambiguous were:

- Combining class, method, constructor, interface and field modifiers
- Cast expressions are allowed to contain expressions as the target object type
- Many expressions take arbitrary expressions as their subexpressions

Each of these had to be caught later in the weeding stage, but it made the grammar construction significantly easier. And the simplified grammar leads to a simpler parse tree which benefited us in AST construction as well.

2.3 AST Construction

AST construction was the most complex component in this assignment, since we had to deal with many different structures in the grammar despite sometimes we just flatted some parts of the concrete syntax tree. I think there are only have two challenges in this part:

- Define abstract enough structures to capture all information of the concrete syntax tree
- Append the token info(file name, line number, column number) to the AST for error reporting of the weeding phase

For the first one, we incrementally changed our abstract structures during implementing those construction functions. We clustered all terminals and nonterminals into several groups in terms of their functions, dependencies and structures in the grammar, and represented each group as an unified well-organized data structures in our implementation. For the second one, we define some "-info" structures with respect to abstract structures to maintain those information.

2.4 Weeding

Weeding was fairly straight-forward, except for catching errors due to excess parentheses. For example, the public tests where unary negation is applied to a parenthesized number that is too big "-(2147483648)", and the public test where a cast was encased in double parentheses "((Object))null".

Those cases were made tricky because during AST generation we flatten expressions as much as possible to create an extremely simple tree. Unfortunately, this means that the outer parentheses,

while still distinguish an expression from valid to invalid, was being discarded for no semantic significance. To circumvent this we added a 'depth' attribute to AST values which is an integer reflecting how far the concrete node it represents was from the top of the tree. The weeder can then look at the depths of AST nodes to determine if statements were excessively nested or not, and the AST is still as flat as possible.

3 Testing

At the moment, our testing strategy is to focus on the provided public marmoset tests. Those tests were ran through each stage of our compiler development.

For example, while developing the scanner, we created *SMain.hs* which test the latest scanner with each of the valid test files, and print out the section of file causing failure to help with debugging. After succeeding on the valid test files, we also included some of the invalid test files to ensure true negatives as well.

The same design was adopted during parser development and AST development. For the close tie between the two stage, the testing was done in the same file, *PMain.hs* which makes sure that all of the valid files pass the parser and AST construction.

By running these front-ends to our compiler we can easily identify failing tests without resubmitting to marmoset. If we fail a test it's also easy to have the test front-end print the parse tree or input string at the time of failure which makes it very easy to resolve issues quickly.