Project Milestone 1

Kevin Ayuque (kja306), Goktug Saatcioglu (gs2417)

February 2019

1 Introduction

This is the documentation for Milestone 1 of the Spring 2019 Compiler Construction course. After Prof. Rose announced that we can work in pairs we decided to work together and split the work as can be seen in Section 3. Other than this file we are submitting a hacs file called `Pr1GoktugKevin.hx` that has our working parser, the folder `/samples/` that has the test files given out with the assignment and the folder `/custom_samples/` that has our own test cases. The choices made and issues resolved is outlined in Section 2. Our explanation of the test cases is available in Section 4. To compile the parse, navigate to the folder containing `Pr1GoktugKevin.hx` and run the following command:

```
> $HOME/.hacs/bin/hacs Pr1GoktugKevin.hx
```

To run a custom test case execute the following command:

```
> ./Pr1GoktugKevin.run --sort=Program custom_samples/FILE_NAME.EXT
```

To run an assignment test case execute the following command:

```
> ./Pr1GoktugKevin.run --sort=Program samples/FILE_NAME.EXT
```

All commands should be executed in a terminal session. We are submitting the same code and same documentation for both of us where the log in Section 3 indicates what was done by whom.

2 Choices Made and Issues Resolved

We've had to resolve a few issues along with making some choices when completing the task required for Project Milestone 1. They are listed below.

1. Starting with **1.1 Definition** (tokens) we see that the requirements for *Identifier* tokens is a little vague. It reads as follows: "*Identifier* tokens start with a letter, $, or _, followed by more of the same as well as digits . . . ". We took this to mean that an *Identifier* can start with a *ValidStart*

and then continue with more characters from either *ValidStart* or *Digit*. As a regular expression, we get

$$\mathit{Identifier} = [a - zA - Z\$\_]([a - zA - Z\$\_0 - 9])^*$$

This is consistent with the way identifiers in C works.

2. In **1.1 Definition** (tokens) the requirements for *Identifier* tokens continues with "...except that keywords (literal tokens used by the grammar) are not permitted)." This issue is resolved by making sure every token of the language is defined according to the assignment paper so no special treatment is necessary.

3. Again, in definition **1.1 Definition** (tokens) comments are either of the form `//` or `/*...*/` where the former is a single line comment and the latter is a multi-line comment. For multi-line comments it is not stated whether comment nesting is allowed (i.e. `/*comment /*comment inner*/ outer*/`). In C nesting of comments is not permitted and we decided to also not permit nesting of comments. The idea was to follow C as closely as possible and also make the work easier for our parser as allowing nesting of multi-line comments would then have to have us use a context-free grammar instead of a regular expression. (The reason why a regular expression does not work for nested comments is because regular expressions cannot count. For example, the language of balanced parenthesis is not regular.) As a final remark, this is obviously not an issue for single line comments as they only need to begin with `//` and end with the newline character.

4. We move onto **1.2 Definition** (expressions) where the order of precedence of certain operations are given. We noticed that there were 9 total levels (lines) of expressions so we chose to use @9 for the highest level of precedence and @1 for the lowest level of precedence. Furthermore, we incorporate left-associative and right-associative operators by following the HACS guide.

5. There is a issue with **1.2 Definition** (expressions) and it has to do with unary operators. In C operations such as `-+5` and `**a` are allowed. The first should evaluate to -5 while the second describes a pointer to a pointer. Since C allows such expressions and it is reasonable to allow them in MiniC too, we decided to make the unary operators right recursive.

6. In **1.2 Definition** (expressions) there is a slight issue regarding how to have a grammar for expression lists. The easy and standard solution to resolve this is to create a grammar that at first either generates empty or a single expression followed by another grammar that either prepends commas or generates empty. This way we can correctly cover all cases for a list of expressions. Our approach also generalizes well to type lists

and statement lists as we can simply use the same grammar with different sorts. Thus, we implement this approach multiple times throughout the parser.

7. Similarly to **1.2 Definition** (expressions), for **1.3 Definition** (types) we have 3 total levels (lines) of expressions so we chose to use @3 for the highest level of precedence and @1 for the lowers level of precedence. Furthermore, parameter type lists are resolved using a similar grammar to that of expression type lists where we again first generate a single parameter and then generate more separated by a leading comma if necessary.

8. The first issue relating to **1.4 Definition** (statements) is that the left side of an assignment must be an l-value. So the left side of the assignment can either be an *Identifier* or a pointer dereference **Expression.* We decided at first to create a new sort called *Lval* that is either an *Identifier* or a **Expression.* This means that we the requirement to have only l-values at the left side of an assignment is in a way hard-coded into the parser. This approach, however, was wrong since now the parser would accept statements such as

   ```
   *(2+2) = atoi(in);
   ```

   which is clearly problematic. So instead we defined *Lval* as either an *Identifier* or a ** Deref* where ** Deref* is equal to either *Identifier* or ** Deref* meaning we can deference multi-directional pointers. This approach works because in C an l-value is either an identifier or a pointer dereference where the pointer dereference is of the form of some amount dereference symbols followed by another identifier. While this may not be the best approach as checking for this during the semantic analyzer is also a possibility, we decided this approach was best in our case in order to pass the test cases and meet the project specifications. As a final note, we did not recursively reference back to an l-value as it might be useful for the future to separate the nodes of l-values in the parse tree which are just identifiers from those that are pointer dereferences.

9. The second issue relating to **1.4 Definition** (statements) is that the grammar given in its current form would lead to the dangling else problem. We decided to resolve using the standard technique of having a new production that eagerly consumes an else when if-then-else statements are generated. The solution was pretty standard but it was important to notice that this was an issue.

10. Statement lists in **1.4 Definition** (statements) were resolved in the same manner that expressions lists and type lists were resolved.

11. For **1.5 Definition** (declaration) there was again the issue of parameter lists which could be easily be solved using the same approach as for ex-

pressions lists, type lists and statement lists. Another issue is that of the `main` function which we discuss in the following point.

12. We see that for **1.6 Definition** (program) we must have a *main* function. This was the most challenging part of the assignment as we couldn't initially figure out how to achieve. Some ideas included creating a new token called *Main* and having it match *main.* This did not work too well because all names that matches to *Main* are also going to match to *Identifier* which confused HACS. Our next approach was to create a new declaration called `MainDecl` and instead of using *Type* and *Identifier* we would just write out *function int main* and follow it up with an optional parameter list. This also did not work since all the tokens were already parsed to either *Type* or *Identifier* making the sort useless and HACS giving a "unreachable statement" error which lead to the program not compiling. Finally, we realized that the order in which the tokens are declared is actually significant for HACS. This means that if a program string has two tokens it matches to, then the very first token declared in the parser source file will be used. Thus, we decided to create a top-level token called *Main* which only matches to *function int main* and then we created a new sort called *MainDecl* that matches to this token. This way we can get the parser to detect whether a `main` function was declared. Furthermore, this also means that we make the following rule on the program: there must exist a `main` function and the `main` function must return an int but it can take any amount of optional parameters. This is the best and most simple solution we came up with. Finally, we defined the program to be a set of optional declarations followed by a `main` declaration followed by another set of optional declarations meaning the parser can enforce whether there is a main function or not. This analysis could have also been left to the semantic analyzer but we chose to hard-code it in in order to pass the test cases. If at a later point we find out that main must take zero arguments then we can easily modify the parser to get the correct behavior but for now allowing optional parameters seems like a good idea as this is also allowed for C like `main` functions (as long as the program is started correctly, i.e. *int main(int argc, char *argv[])* is a valid C main function and needs to be called with command line arguments).

3 Work log

Date	Details	Description
February 11	Kevin Ayuque Goktug Saatcioglu	Team formation.
February 12	Kevin will work on tokens and types. Goktug will work expressions and statements. Agreed to use GitHub and Slack. Agreed to a common naming scheme. Will resolve issues as they come up Meet up Friday	First goals set. Work divided up.
February 15	Realized certain issues. First issue if of main. Another issue is of l-values. Another issue is of dangling else. Worked together to fix the issues. Got all test cases to pass. Pushed final version to Git. Divided work for the writeup. Goktug will write Section 2. Kevin will write Section 4. Kevin will add custom tests.	Second meeting. Finished parser. Divided up work.
February 20, 21	Certain bugs were resolved. Documentation finished	Testing and Documentation.
February 22	Finished documentation. Discussed implementation one last time. Wrote Introduction (Section 1).	Final meeting for part 1.

4 Testing

4.1 Sample tests

```
> ./Pr1GoktugKevin.run --sort=Program samples/comments.badMC
```

```
1  /* This comment
2     /* Has a nested one */
3  */
4
5  function int main() {
6    return 0;
7  }
```

The above does not compile because comments do not nest. The only comments ignored are from line 1 to line 2. On line 3 the compiler is expecting a sequence of declarations or a comment to ignore, but `*/` does not match any of the criteria.

```
> ./Pr1GoktugKevin.run --sort=Program samples/comments.MC
```

```
1   // Some comments.
2
3   /*****
4     Wrapper comment
5     // single line comment
6   *****/
7
8   // Single with /* wrapped */
9
10  function int main() { return 0; }
```

The above does compile. There is a comment on line 1, a multi-line comment from line 2 to 6 and a comment on line 8 that are ignored, followed by the `main` declaration.

```
> ./Pr1GoktugKevin.run --sort=Program samples/doubleelse.badMC
```

```
1   function int main(*char in) {
2     if (1==2) {} else {} else {}
3     return 0;
4   }
```

The above does not compile because there is an additional `else` on line 2 and `else` is not at the beginning of any statement.

```
> ./Pr1GoktugKevin.run --sort=Program samples/nomain.badMC
```

```
1   function int f(*char in) {
2     return 0;
3   }
```

The above does not compile because the program expects a sequence of declarations with at least one `main` function, which is not the case here.

```
> ./Pr1GoktugKevin.run --sort=Program samples/nonlvalue.badMC
```

```
1   function int main(*char in) {
2     2+2 = atoi(in);
3     return 0;
4   }
```

The above does not compile because on variable assignment we limit the left expression to be an identifier or a pointer dereference. Therefore `2+2` on the left side of the expression is not allowed.

```
> ./Pr1GoktugKevin.run --sort=Program samples/strcpy.MC
```

```
1  // Copy a string in Mini-*-C-*-.
2  function *char strcpy(*char string) {
3    var int length;
4    length = strlen(string);
5    var *char copy;
6    copy = malloc(length+1);
7    var *char p;
8    p = copy;
9    while (*string) {
10     *p = *string;
11     string = string + 1;
12     p = copy + 1;
13   }
14   *p = 0;
15   return copy;
16 }
17
18 function int main(*char input) {
19   var int dummy;
20   dummy = puts("The copy of the string is ");
21   dummy = puts(strcpy(input));
22   return 0;
23 }
```

The above does compile. There is a comment on line 1 that is ignored, followed by a sequence of declarations that includes the `main` function.

```
> ./Pr1GoktugKevin.run --sort=Program samples/strings.MC
```

```
1  // Strings in Mini-*-C-*-.
2  function int main() {
3    var int dummy;
4    dummy = puts("This string has a \" and \n and \
5  then it continues with \x68ex and \t\ttabs");
6    return 0;
7  }
```

The above does compile. There is a comment on line 1 that is ignored, followed by the `main` declaration that includes 3 valid statements.

```
> ./Pr1GoktugKevin.run --sort=Program samples/strlen.MC
```

```
1  // Compute string length in Mini-*-C-*-.
2  function int strlen(*char string) {
3    var int length;
4    length = 0;
5    while (*string) {
```

```
6       length = length + 1;
7       string = string + 1;
8     }
9     return length;
10  }
11
12  function int main(*char input) {
13    var int dummy;
14    dummy = puts("The length of the string is ");
15    dummy = puti(strlen(input));
16    return 0;
17  }
```

The above does compile. There is a comment on lin 1 that is ignored, followed by a sequence of declarations that includes the `main` function.

4.2 Additional tests

```
1   // This method returns the nth Fibonacci number
2   function int fibonacci(int input){
3       var int i1;
4       var int i2;
5
6       i1 = 1;
7       i2 = 1;
8
9       while(input > 2){
10        var int temp;
11        temp = i2;
12        i2 = i1 + i2;
13        i1 = i2;
14        input = input - 1;
15      }
16      return i2;
17  }
18
19  function int main(*int input) {
20    var int tenth;
21    tenth = fibonacci(10);
22    return 0;
23
24  }
```

The above does compile. There is a comment on line 1 that is ignored, followed by a sequence of declarations that includes the `main` function. The `fibonacci` and `main` functions both contain valid statements.

```
1  // This method returns the nth Fibonacci number
2  function int fibonacci(int n){
3      var int i1 = 1;
4      var int i2 = 1;
5
6      while(n > 2){
7        var int temp = i2;
8        i2 = i1 + i2;
9        i1 = i2;
10       n = n - 1;
11     }
12     return i2;
13 }
14
15 function int main(*int input) {
16   var int 10f;
17   10f = fibonacci(10);
18   return 0;
19
20 }
```

The above does not compile. Unlike the previous example, on line 3,4 and 7, our compiler does not allow variable declaration and variable assignment on a single statement.

```
1  // This is a very inefficient implementation of pow
2  function int pow(int base, int exponent){
3      if (exponent == 0){
4          return 1;
5      }
6      var int res;
7      res = base;
8      while(exponent > 1){
9        res = res * base;
10       exponent = exponent - 1;
11     }
12     return res;
13 }
14
15 function int main(*int input) {
16   var int res;
17   res = pow(10, 2);
18   return 0;
19 }
```

The above does compile. There is a comment on line 1 that is ignored, followed by a sequence of declarations that includes the `main` function. The `pow` and `main` functions both contain valid statements.