Release Report

INF01147 - Compilers

Henrique Corrêa Pereira da Silva and Bernardo Hummes Flores

Informatics Institute
Universidade Federal do Rio Grande do Sul

{hcpsilva, bhflores}@inf.ufrgs.br

September 20, 2020

Abstract

After a non-optimal first stage, we implement a fully-featured syntax parser, as defined in the second stage of the assignment. The parser conforms to the given grammar rules and also implements rich error output through several features of both `flex` and `bison`. As future work, we intend to perpetuate the modular design and extend the source code documentation.

1 Introduction

In this document we will focus on this course's second objective of the final assignment: a syntax parser. We shall touch upon the definition itself in Section 2, talk about the tools utilized in Section 3 and the conclusion in Section 4.

2 Objective

The second stage outlined a parser that implements a language with the following building blocks:

- Variable assignment;
- Function definition;
- Command blocks;
- Simple commands;
- Arithmetical and logical expressions.

All this would be enabled by `bison`, the GNU's `yacc` implementation. With both `flex` and it, we had little trouble to implement all that was put up in this milestone.

3 Tools

By the first look upon the `bison` instruction manual, we noted that it was indeed a well-featured tool. From the get-go we turned on all warnings and reports, the latter which proved to be specially useful when trying to find ambiguities in the generated grammar. That being said, some assembly was required in order to elegantly generate the header file in the correct location, following our project structure.

After some struggle, we decided to enhance the error printing through the addition of some context: the whole error line. In order to do this, one must enable the `%location` flag in `bison`:

But, well, that's far from all we had to do. After that, we added the input line buffer to `flex`:

```
/* to track the initial column of matched tokens */
int yycolumn = 1;
/* to save the line of the current matches*/
char* yylinebuf;
int yylinebuf_len = 0;

/* helpful flex feature that helps us to track the location of the tokens */
#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; yylloc.last_column = yycolumn + yyleng - 1; \
    yycolumn += yyleng;
```

The last define allow us to maintain the existing `lex` action mostly unchanged. That's because we had to actually copy the contents of the input to that buffer:

```
/* if our line buffer is small, we grow it by 100 characters */
if (yyleng + 1 > yylinebuf_len) {
    yylinebuf = realloc(yylinebuf, (yyleng+100)*sizeof(char));
    yylinebuf_len = yyleng + 100;
}
strncpy(yylinebuf, yytext, yylinebuf_len - 1); /* in the <NORMAL>\n.*
                                                * rule we copy from
                                                * yytext+1 onwards */
yycolumn = 1;
yyless(0);
BEGIN(NORMAL);
```

That's the action of two rules: `.*` and `<NORMAL>\n.*`. The state `NORMAL` had to be used in order to correctly match the whole first line of input. That is, the standard `INITIAL` state is only used to copy the first line of input. Perhaps not that performant, but it's only done once.

With that information, we finally implement the `yyerror` function:

```
void yyerror(char const* s, ...)
{
    va_list ap;
    va_start(ap, s);
    char* underline = cc_parser_underline(yylloc.last_column + 2,
                                          yylloc.first_column,
                                          yylloc.last_column);
```

```
8
9         if (yylloc.first_line) {
10            fprintf(stderr, "%d:%d: error: ",
11                    yylloc.first_line,
12                    yylloc.first_column);
13        }
14        vfprintf(stderr, s, ap);
15        fprintf(stderr, "\n    | %s\n    | %s\n",
16                yylinebuf, underline);
17
18        free(underline);
19    }
```

Why variadic parameters? Well, maybe we need to actively print out errors with more information in our grammar rules. Anyways, the output is the following:

```
7:13: error: syntax error, unexpected ',', expecting ':'
    |   for(i <= 0, j <= 0 : i : i <= 2, j <= 3) {
    |             ^
```

With that done, we again did implement a test suite of sorts in the shape of a shell script (included together with this release). Here is a snippet of it:

```
----------------------------------------------------------------
TEST CASE OF FILE 'ccompiler/test/etapa2/teste_065':

7:13: error: syntax error, unexpected ',', expecting ':'
    |   for(i <= 0, j <= 0 : i : i <= 2, j <= 3) {
    |             ^

FILE CONTENTS:

//ERROR: As construções iterativas são separadas com ':' e só uma atribuição
int f()
{
  int i;
  int j;
  for(i <= 0, j <= 0 : i : i <= 2, j <= 3) {
  };
}
```

4 Conclusion

The choices made early in this assignment paid out: we had little trouble with this milestone, and the source code is still well modularized. We hope that it keeps paying out throughout the milestones, as we expect the code complexity to increase considerably.