Module 1
Lexical Analysis, Syntax Analysis, and Semantic Analysis

Compilers and Execution Environments (ID2202)
7.5 hp

David Broman
KTH Royal Institute of Technology

Monday 8th November, 2021

Abstract

Welcome to Module 1 of the course! In this module you will learn how to construct a scanner (lexical analysis), how to parse files (syntax analysis), how to interpret expressions, and how to perform name binding checks as well as basic type checking (semantic analysis). The goal of Module 1 is that you have a front-end for parsing a subset of or the complete Cigrid language. Besides the practical implementation tasks, you will also solve a number of individual theoretical exercises.

This document outlines the different exercises for Module 1. The exercises are divided into three levels: Satisfactory (S), Good (G), and Very Good (VG). Depending on your goal of the final grade in the course, you may choose which exercises that you perform. Please note that the minimum for passing the course is to complete all exercises at level Satisfactory (S). To reach level (G) for this module, all exercises at level (S) and (G) have to be completed. Finally, to achieve level (VG), you need to finish all exercises at all three levels. For details about the grading, see the course memo on Canvas.

Contents

1 Level: Satisfactory (S):

The goals of the tasks at level Satisfactory (S) are to learn about context-free grammars, top-down parsing using recursive descent, interpretation of simple expressions, and using a parser generator. Level (S) consists of three parts.

1.1 Lexing, parsing, and interpretation of a calculator

In this exercise, you will create your own recursive descent parser for a simple calculator expression language.

Recommended Reading

Before you start, we recommend that you

- listen through and reflect on Lectures 4, 5, and 6,
- take a closer look at the examples about the calculator in Lecture 5, and
- read section 3.1-3.3 in the course book (Cooper & Torczon, 2012).

Task

Your task is to create an interpreter for a small calculator language. You should store your solution in folder `solutions/module1/calc/`. Within this folder, it should be possible to compile the program using the `make` command. The compiled program must be called `calc` and stored in the folder given above. When executing the program, the calculator should read from standard input, one line at a time. For each line, it should parse the expression, print out the expression with explicit parenthesis, compute the resulting value, and print it out on a new line. If the program reads an empty line, it should terminate. For instance, if a user starts the program by running `./calc` from the command line, and enters the following expression

```
1+2+3*4
```

and hits return, the program should print the following to the standard output:

```
((1 + 2) + (3 * 4))
= 15
```

Then, if the user enters another expression, the process is repeated. Note how normal precedence rules have been taken care of in the correct way.

The calculator should handle the following ambiguous grammar:

$$
\begin{aligned}
expr &\rightarrow expr \texttt{ "+" } expr && (1)\\
&\mid expr \texttt{ "-" } expr && (2)\\
&\mid expr \texttt{ "*" } expr && (3)\\
&\mid expr \texttt{ "/" } expr && (4)\\
&\mid \texttt{Num} && (5)\\
&\mid \texttt{"(" } expr \texttt{ ")"} && (6)
\end{aligned}
$$

associativity定义：用来形容运算符号
left associative operator: 将视角放在此operator上，左手边的expr已经算好了，右手边的那一大堆剩下的token，读到哪里才算作此operator的右手边expr呢：读到第一个优先级<=此operator的operator为止。比如1*2^3*6*8+9，读到3
right associative operator: 将视角放在此operator上，左手边的expr已经算好了，右手边的那一大堆剩下的token，读到哪里才算作此operator的右手边expr呢：读到第一个优先级<此operator的operator为止。比如1*2^3*6*8+9，读到8

precidence定义：precidence其实是用来形容production的，而不是用来形容运算符号的

Left associative because recursion on the left term of the "+"
(参见Lec5 slide 12)

1. eliminate the ambiguaty. aka declare rules of precedence and associativity:

```
expr -> expr "+" term
      | expr "-" term
      | term

term -> term "*" factor
      | term "/" factor
      | factor

factor->Num
      | "(" expr ")"
```

2. eliminate left recursion

```
expr -> term expr'
expr'-> "+" term expr'
      | "-" term expr'
      |

term -> factor term'
term'-> "*" factor term'
      | "/" factor term'
      |

factor-> Num
      | "(" expr ")"
```

expr', term'用来:
输入已有左半边个体，识别新连接符，识别右半边个体，连接左右半边个体形成新个体，将新个体作为新的左半边个体，传递给下一个expr'或term'

expr, term, factor用来：
生成完整个体（expr对应+-粘合而成的个体，term对应*/粘合而成的个体，factor对应数字或()个体），然后将此个体传递给expr'或term'这两位连接器

```
Left association:
expr -> expr "+" term
      | expr "-" term
      | term

term -> term "*" factor
      | term "/" factor
      | factor

factor->Num
      | "(" expr ")"
```

```
Right associative:
expr -> term "+" expr
      | term

term -> factor "*" term
      | factor

factor->Num
      | "(" expr ")"
```

只要遇见的运算符是"+"及以上的，AST就一直向右扩张生长
换一种叙述方式就是：
AST一直向右扩张生长，直到遇见的运算符低于"+"

只要遇见的运算符是"*"及以上的，AST就一直向右扩张生长
换一种叙述方式就是：
AST一直向右扩张生长，直到遇见的运算符等于或低于"+"

To make the grammar unambiguous, we assume the standard precedence and associativity rules. That is, all binary operators are left associative. Operators `*` and `/` have the same precedence and they have higher precedence than operators `+` and `-`. Operators `+` and `-` have the same precedence. The terminal symbol `Num` represents an unsigned integer. In your scanner, treat space, tabs, and newline characters as whitespace.

For instance, assume you have a file `example.txt` with the following content:

```
1 + 10 / 3
(25 - 10) * 3
```

If you execute the following line

```
cat example.txt | ./calc > out.txt
```

then file `out.txt` should contain the following text:

```
(1 + (10 / 3))
= 4

((25 - 10) * 3)
= 45
```

中间空几行都行

Note that you must implement your parser as a recursive descent parser. For (S) level, you must NOT use precedence climbing (which is an add-on task for (G) level). To be able to pass all unit tests, the pretty printer must use the same spacing between literals as in the example above, as well as printing parenthesis around all expressions, except the integer literals. The number of empty lines between expressions does not matter.

The program must return the following error return codes:

- Error code 0 if there are no lexing, parsing, or execution errors.
- Error code 1 if there is a lexing or parsing error.
- Error code 2 if there is a runtime error during execution.

Grading tag

Use the following tag when submitting this exercise: `#module1-calc-s`

1.2 Scanning, parsing, and pretty printing Cigrid programs

In this exercise, you will create a parser using a parser generator for a subset of the Cigrid language.

Recommended Reading

Before you start, we recommend that you

- listen through and reflect on Lectures 3, 4, and 5,
- read through the whole Cigrid language reference manual, and
- read section 2.1-2.2, 2.5, 3.1-3.5 in the course book (Cooper & Torczon, 2012).

Task

Implement the first version of your general Cigrid compiler, which includes a scanner, a parser, and a pretty printer. In the following exercises (for higher grades) and coming modules, you will extend this implementation with more features.

You need to use an approved parser generator tool that is using a bottom-up parsing strategy. Examples of approved parser generator tools are the tools `ocamlyacc` or `Menhir` for OCaml, the `Happy` tool for Haskell, and `LALRPOP` for Rust. If you are implementing your compiler in some other language, please contact David Broman, `dbro@kth.se` and give a suggestion of the parser generator that you would like to use.

Locate all the code of your Cigrid compiler in the folder `solutions/cigrid/`. The compiler must be possible to compile by typing `make` in the above folder. After compilation, an executable file called `cigrid` should be available in this folder.

Your program should be able to take one argument, which is the file to be compiled. In the coming exercises, you will add several parameters to the program. Hence, it might be a good idea to use an argument parsing library for parsing program arguments, but it is not a requirement.

If the argument `--pretty-print` is given, the AST of the parsed Cigrid program should be pretty printed according to the Cigrid Language Reference Manual, Section 2.3. Note that it is important to follow the grammar for pretty printing, but you do not have to do any indentation etc. if you do not want to. The examination system ignores whitespace.

Note that when the grading system is calling your Cigrid compiler from the command line, all flags are given before the filename. Note also that the order of the flags should not matter. For instance, `cigrid --pretty-print examples.cpp` is a valid command line.

If the compiler is invoked with an incorrect flag, the compiler must return error code 1, and preferably print out a command menu (the format of the menu is not checked by the grading system).

In this task, lexing and parsing of all parts of the Cigrid language need to be handled, except the following parts that are optional:

- Hexadecimal integers
- Shift operators >> and <<
- Unary operators `~`, `-`, and `!`
- Identifiers in types and pointer types
- String constants
- The `new` and `delete` constructs
- Array access constructs
- Array assignments
- The `struct` construct (both global definitions and array access / assignments)
- The `for` construct

- Global variables (both definitions and declarations)

Note: If there is an error in the scanning and parsing, error code 1 must be returned, else error code 0. Some advice:

- Start by developing the scanner and test the different tokens individually.
- While you develop your parser, test small parts of the file `examples.cpp` (available on the Canvas page). Simply comment out the parts that you do not want to test.
- To be able to parse the example file, you need to ignore pre-processing directives (for instance `#include`). The simplest way to achieve this is to treat the character `#` in the same way as a single line comment.
- The operators `++` and `--` should be implemented as syntactic sugar.

Note that a Cigrid program should always be possible to compile using a C++ compiler. However, some compilers, such as GCC or Clang, give warnings for certain programs that are legal Cigrid programs. Hence, a simple way to test a Cigrid program using GCC is to create a shell script and to disable such warnings. To do this, create a file called `cigrid-gcc`, enable execute access using command `chmod +x cigrid-gcc`, and include the following text in the file:

```
#!/bin/bash
g++ "$@" -Wno-dangling-else \
-Wno-c++11-compat-deprecated-writable-strings
```

Note that it is not necessary to achieve more than S on module 1 to be able to get higher grades on module 2 and 3. However, to solve the tasks for higher grades on module 2 or 3, your Cigrid parser must handle the language parts required for level G in module 1 (Section 2.1). If you later decide to get higher grades on module 2 or 3, it is OK to extend the parser at that time.

Grading tag

Use the following tag when submitting this exercise: `#module1-cigrid-s`. Note that this tag also includes tests for the next section (name analysis).

1.3 Name Analysis

In this exercise, you will create a simple name analysis.

Recommended Reading

Before you start, we recommend that you

- listen through and reflect on Lectures 6.
- read through the whole Cigrid language reference manual (especially Section 3 on semantics), and
- read section 4.1 - 4.2 in the course book (Cooper & Torczon, 2012).

Task

In this task you should implement a function that traverses the AST and performs simple name analysis, a form of semantic analysis. The name analysis should be enabled if flag `--name-analysis` is given. The semantic function should check the AST to make sure that all variables that are used inside a function either are defined as function parameters, or as local variables. Checking use of global variables, of types, or that function names are defined before use are not parts of S level (this is part of the VG level). If the analysis discovers that a name has not been defined before its use, the program should terminate with return code 2.

Grading tag

Use the following tag when submitting all exercise for Cigrid at level S:
`#module1-cigrid-s`.

1.4 Individual theoretical exercise

Within a few days of start of the module, a number of theoretical examination tasks will be available in your Git repository under folder `exercises/module1/`. These tasks are individual, that is, each student in the course gets an individual set of exercises.

You are NOT allowed to talk to anyone about these exercises or to get any help from anyone to solve them. You are not allowed to publish or distribute your solutions or the exercises. You must write down your solution using pen and paper and then take a photo of the solution. For the exercises marked as satisfactory, submit your solution on Canvas (as one or several image files). You can find the submission page under "Assignments" on Canvas. Note that you might be asked to explain your solution (in detail) at the seminar.

2 Level: Good (G):

The goals of the tasks at level Good (G) are to get a deeper understanding of parsing techniques, both theoretically and practically.

2.1 Complete the Cigrid parsing

In this task, you should complete the Cigrid parser by extending your parser from (S) level. Specifically, you should:

- complete the lexing, parsing, and pretty printing of the whole Cigrid language specification,
- include an error message that prints out a precise line number (to standard error) for lexing and parsing errors,

If there is an error in the scanning and parsing, error code 1 must be returned, else error code 0. You should continue to implement on the same code base as previously described in Section 1.2. That is, your Git repository should only contain one `cigrid` folder, that is valid for all exercises.

In this exercise, you should add a new program parameter `--line-error`. If this parameter is enabled, the compiler should report a lexing or parsing error by *only* printing out the exact line number where the error occurs (as a decimal number followed by a new line character). Note that the error should be printed to *standard error* and not to standard output.

Grading tag

Use the following tag when submitting this exercise: `#module1-cigrid-s-g`. Note that it will check that the Cigrid implementation still works for level (S).

2.2 Precedence Climbing

In this task, you should learn about precedence climbing for top-down parsers. Specifically, extend the calculator from (S) level to support precedence climbing. Your existing solution (without precedence climbing) should be kept. Hence, if flag `--precedence-climbing` is given, the new code should be invoked. If the flag is not present, the old code should be executed. Besides the operators explained in the level (S) exercise, the new implementation should also include an infix power operator `^`. For instance, an expression `2^3` results in a value 8. Note that the `^` operator must have the highest precedence and be right associative.

```
unary operator = {prefix, postfix}
binary operator = {infix}
```

Grading tag

Use the following tag when submitting this exercise: `#module1-calc-s-g`.

2.3 Individual theoretical exercise

See Section 1.4 for information about individual theoretical exercises, and how you submit on Canvas. Note that there are different submission pages for the two different levels S and G.

3 Level: Very Good (VG):

The goal of this task is to extend the Cigrid compiler with preprocessing directives, good error reporting, as well as performing type checking of a Cigrid program.

3.1 Include preprocessor directives and error reporting

In this task, you should extend your Cigrid compiler to support the `#include` C preprocessing directive, following this grammar

$$\textit{preproccing} \rightarrow \texttt{"\#"}\ \texttt{"include"}\ \texttt{TextStr} \tag{7}$$

where the syntax of the grammar is defined in the Cigrid language reference manual (see Section 2.3 of the manual for information about `TextStr`). If such include construct is located in a Cigrid file, the text from the include file should replace the `#include` statement line. Note that when the text is inserted, it should also insert new line characters, both before and after the included text. It is not necessary to recursively interpret preprocessing directives in an included file, but it is allowed to do so.

This preprocessing can be done using the GNU `cpp` command, i.e., that the Cigrid compiler invokes the `cpp` command automatically before parsing. An alternative is to perform the inclusion of included files on the fly, during parsing. Both approaches are allowed. However, the following two requirements must be fulfilled:

1. The compiler needs to give good line error reporting (to standard error), reporting parsing errors where at least the line number and the file name is explicitly printed out. This error reporting should be done if program parameter `--line-error` is NOT used. If the above parameter is used, only the line number should be printed out, as described in the exercise for level G.

2. File inclusion that is using an include directive with angle brackets must be ignored, that is, the file is NOT included. For instance, a line `#include <stdio.h>` should be replaced by an empty line.

Note that fulfilling the above two requirements using the `cpp` command can be challenging.

3.2 Type checking and more name analysis

```
Checking use of global variables, of types, or that
function names are defined
before use
```

In this task, you should implement a very simple type checker and more comprehensive name checks. The checker needs to reject programs according to what is specified in the Cigrid specification. Please see the section about static semantics. Note that checking return types are optional (not needed to pass the tests). These checks should only be performed if flag `--type-check` is enabled.

If there is an error, return code 2 should be returned. Also, if `--line-error` is given, the correct line number should be printed (in the same way as for the exercise at level (G)).

Grading tag

Use the following tag when submitting this exercise: `#module1-cigrid-s-g-vg`. Note that it will check that the Cigrid implementation still works for level (S) and (G). This tag is used for both the type checking and for preprocessing. Some tests will also be performed manually at the seminar. Note also that to pass VG level, tag `#module1-calc-s-g` must still work.