CS 131 Compilers: Discussion 1: Lexical Analysis

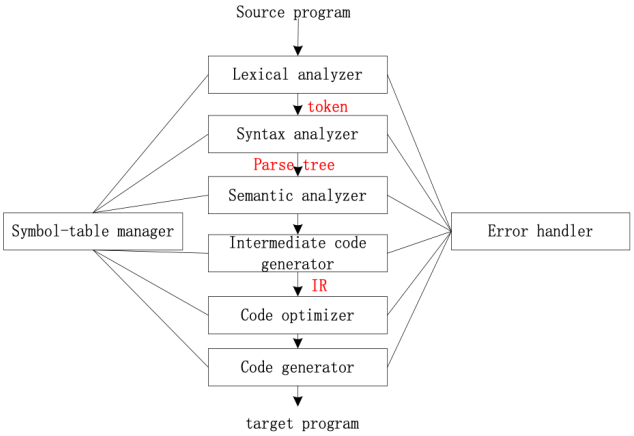
杨易为 吴凌云 樊雨鑫
{yangyw,wuly2,fanyx}@shanghaitech.edu.cn

2022 年 2 月 23 日

1 Welcome to the Compiler's World

1.1 Some General Views

We've covered on class that the compiler includes lexical analysis, in which we get tokens. This process is called scanner of the program.



To make detail definition of compiling a program into a machine recognizable text and discuss whether kind of language is "readable" by a Turing Machine or so-called computer, we introduce regular expressions. Formally, we have kleene closure and possitive closure of a language to describe a language. Then NFA, DFA, DFA minimization are tools to accept the language. Regex have some limitation of describing a language and how to evaluate the power of it? Context Free Grammer is introduced and its solution LL Parsing and LR Parsing.

Till now, we have made the program into a Intermediate Representations, which in our project is Abstract Syntax Tree. We'll then make some type checking on it. This takes into the definition of Operational Semmantics. For modern language like LLVM or those who compiled into LLVM IR, they have a Intermediate Representations where passes can be cast on to optimize the code. Those optimizations are basically machine unrelated. Other intermediate representations include 3 address code and Single Static Code. Some of them are more low level and others are very similar to the AST, which depends how heavy the analysis and the optimization can cast on. We'll cover intra-basic-block optimization，dataflow analysis，security problems like Control Hijacking Attack. Language may alloacate resources on stack or heap, we can do runtime garbage collections to make the resources reallocatable.

The eventual process of a compiler is Code Generator to generate machine code, we'll cover transform the IR into the riscv backend that we've covered on Computer Architechture classes. Wish you a good luck at Compiler class.

1.2 Why Both Parser and Lexer?

Only lexer in dealing with the C++ code do not deal with nested grammer. The well defined C++ code is even not acceptable by the modern parser.

In the old compiler, We don't recognize *vector < vector <>>*. For example *vector < vector < int >>* v, some compiler will compile into vector ¡ vector ¡ int ¿¿ v. Other than that, we have Most vexing parse.

```
1 struct Timer {};
2 struct TimeKeeper {
3   explicit TimeKeeper(Timer t);
4   int get_time();
5 };
6 int main() {
7   TimeKeeper time_keeper(Timer());
8   return time_keeper.get_time();
9 }
```

TimeKeeper time_keeper(Timer()); is ambiguous, since it could be interpreted either as a variable definition for variable time_keeper of class TimeKeeper, initialized with an anonymous instance of class Timer or a function declaration for a function time_keeper that returns an object of type TimeKeeper and has a single (unnamed) parameter, whose type is a (pointer to a) function[Note 1] taking no input and returning Timer objects.

Julia is language with no Lexer and with parser, its parser reads from the string stream to generate the AST. It makes a lot of Context-Sensitive keyword and grammer struct which is difficult to formally describe, functional language's manual is shorter than languages like C++. But we can see that only parser is actually possible to parse the code. The lexer to decide the token is for sake of speed. But Lex/Yacc is fast enough to take over the speed. In Julia, operator and token are the same. import Base.* means import operator* from class Base, also means import all subclasses in class Base.

Thus mere lexer and mere parser are not wise.

1.3 How can a simple regex work in your project?

1. Any Search with Regex option.
2. You can write a plugin for a new language like Julia to support highlighting in vim/vscode/jetbrains.

1.4 Stages of Compilations

1.4.1 Convert the C++ source code into a sequnce of tokens.

Compilers are conceptually broken down into multiple phases, each of which carry outtransformations on a source program towards the compiler' s ultimate goal; the gener-ation of a target program. The result of each phase is an intermediate representationthat facilitates the implementation of subsequent phases (or, if it is the last phase,the result is the target program). The first phase in a compiler, the lexical analyzerorlexer, reads an input stream of characters and converts it into a stream oftokensto be sent to the parser for syntactic analysis.

```
1 // n! = 1 * 2 * 3 * .... * (n-1) * n
2 int factorial(int n) {
3     int result = 1;
4     for(int i = 2; i <= n; i++)
5         result *= i;
6     return result;
7     }
```

Answer:

1.4.2 The rest of the compilations.

What are the rest of the stages of compilation? List them in order and write a briefdescription about what they and what they take as input and what they output.
Answer:

2 Regular Expressions

The lexer needs to scan and identify finite character sequences that match a pattern corre-sponding to a particular token (the matching strings are also calledlexemes). Patterns offinite strings can be codified usingregular expressions, each of which specify what is calledaregular language. Kleene' s theorem states that any regular language can be recognized bya finite state automaton (FSA) and any language that is recognized by an FSA is regular.

2.1 What language is denoted by each of the following regular expressions?

Try writingdown a few simple strings and give a concise description of the language.

2.1.1 [a-zA-Z][a-zA-Z0-9]$*$

Answer:

2.1.2 $((\epsilon \mid a)b*)*$

Answer:

2.2 Write a regular expression for the following languages.

2.2.1 Uppercase Letters

All strings of uppercase letters, where the letters are in ascending lexicographicorder (empty string allowed).
Answer:

2.2.2 ShanghaiTech Teacher's email addreess

xxxx@shanghaitech.edu.cn, different error handler for different inputs? Suppose you have a large spreadsheet, each row of which has a firstname,lastname,and email column. You want to find all people who use an email address of the form firstname+lastname@shanghaitech.edu.cn. Write a regular expression using backreferences that describes this pattern.

Answer:

2.2.3 Months and Dates

The following four strings: October 8th, October 8, Oct 8th, Oct 8 (Be as conciseas possible with your regex)

Answer:

2.2.4 Ones and Zeros

Even binary numbers without leading 0.

Answer:

2.2.5 Ones and Zeros Plus

Binary sequeance that can be divided by 5 without remainings.

Answer: