Compiler Development

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

ANTLR

- There are many tools to help build a (simple) compiler relatively easily from scratch
- A very popular such tool, which is as easy as it gets, is ANTLR (ANother Tool for Language Recognition)
- What we do in these slides:
  - Define a simple language
  - Build an ANTLR lexer for it
  - Build an ANTLR parser for it
  - Build an ANTLR code generator for it
  - And just like that, we have a compiler!

Installing ANTLR

- This module has a reading “Installing ANTLR”
- Basically, installing ANTLR is just downloading a .jar file
- The above reading has a step-by-step “hello world” of running ANTLR, that we’ll skip for now

Our Source Language

- A very simple language with many things missing
  - We'll grow it, and possibly tweak its syntax, later
- The example on the right shows more-or-less everything in the language
- Let's try to list everything…

```
int a;
int b;
a = 3;
b = a + 1;
if (b == 4)
    a = 2;
endif
if (a == 3)
    a = a + 1;
    b = b + 6;
endif
print a;
print 12;
```

Our Source Language

- Say we want to define a language with the following:
  - Reserved keywords: int, if, endif, print
  - An addition operator: ‘+’
  - An assignment operator: ‘=’
  - An equal operator: ‘==’
  - Integers
  - Variable names as strings of lower-case letters
  - Semicolons for terminating statements
  - Left and right parentheses
  - The ability to ignore white spaces, tabs, carriage returns, etc.

ANTLR .g4 File

- With ANTLR can implement our entire compiler, step-by-step, in a single source file
- Let's call it MyLanguageVx.g4
  - MyLanguageV0 will be the lexer only
  - MyLanguageV1 will add the parser
  - MyLanguageV2 will add code generation
- The structure of the .g4 file is really simple…

ANTLR .g4 File

```
// Grammar name and definition
grammar MyLanguageV0 // For file MyLanguageV0.g4
program :
        EOF
        ;
// Regular Expression definitions
REGEXNAME1: <regex> ;
REGEXNAME2: <regex> ;
// Ignore all white spaces
WS: [ \t\r\n]+ -> skip ;
```

- The above grammar is empty (a grammar definition is required)
  - “program” is the grammar’s non-terminal
  - The “:” is the derivation arrow
  - So the above grammar is really: $S \rightarrow \varepsilon$

ANTLR Regular expressions

- ANTLR uses an easy/standard syntax for regular expressions that we can just learn through examples:
  - DIGIT : [0-9] ;
  - VARIABLE: [a-z]+ ;
  - EQUAL: '==' ;
- The only one that's tricky is the one to ignore white spaces:
  - WS: `[ \t\r\n]+ -> skip ;`
- Let's implement our lexer in file MyLanguageV0.g4…
- The "A Simple ANTLR lexer" reading shows all details
  - Includes a link to a complete MyLanguageV0.g4 file Includes a convenient Makefile (Makefile_ANTLR_x86)
  - The **`-tokens`** flag will print out the lever-generated tokens

ANTLR Grammar Rules

- Again, ANTLR uses a straightforward syntax for grammatical rules
  - A nonterminal symbol (lower case)
  - A ":" that is the derivation arrow →
  - A sequence of non-terminal symbols and of regular expressions, with pipes ("|") used for doing an or
  - A closing ";"
  - One can put new lines and spaces wherever
- Examples:

```
// Production rule for a print statement
printstmt : PRINT term SEMICOLON ;

// Production rule for a term
term :
          NAME
        | INTEGER
          ;
```

Our ANTLR Parser

- There are many options when writing a CFG for a language!
  - In particular, we can pick whatever hierarchy of non-terminal symbols
- Let's get started on writing our CFG in file MyLanguageV1.g4, which will automatically produce a parser, but let's not do the whole thing…

- See details in the "A simple ANTLR parser" reading
  - Let's look at the MyLanguageV1.g4 that's linked off that reading, run it, and use it going forward
  - It has a relatively deep hierarchy of non-terminals, which will seem a bit surprising but is very typical of real-world CFGs
  - the **`-gui`** flag will pop up a GUI that shows the parse tree

Implementing the Backend

- We now have our compiler front-end, with our lexer and parser
  - Whose code is automatically generated by ANTLR
- Let's now implement our backend, which does code generation
  - This is the most interesting part of course

Code Generation

- Code generation is a pretty complex part of compilers, especially because the generated code should be fast
- One easy, but limited option, is to use syntax-directed translation
  - Attach *actions* to the rules of the grammar
  - Use *attributes* to non-terminals and terminals in the grammar
- So we don't really implement a separate back-end, we just augment our front-end so that it generates code
- There is quite a bit of theory here, but instead we'll just do it by example using the ANTLR syntax
  - ANTLR is so easy, that seeing examples is enough!
- First let's just see how one can get ANTLR to output text, based on the rule of our grammar…

ANTLR Syntax-directed translation

- Each time a grammar symbol is evaluated you can insert Java code to be executed!
- Example:

```
program :
    {System.out.println("Declarations!");}
     declaration*
     {System.out.println("Statement!");}
     statement*
    {System.out.println("Done!");}
     ;
```

ANTLR Syntax-directed translation

- Let's start from the MyLanguageV1.g4 file on the course Web site and copy it into MyLanguageV2.g4 (changing the grammar's name in it as well)

- Let's add a tiny bit of Java to our parser to generate the standard parts of an x86 NASM program as we've done by hand this semester: preamble, cleanup, etc.

ANTLR Syntax-directed translation

- Each (lexer) token has an attribute called `text` that contains its lexeme
- Example:

```
declaration :
    INT NAME SEMICOLON
    {System.out.println("Declared "+$NAME.text);}
    ;
```

ANTLR Syntax-directed translation

- Let's add more Java to MyLanguageV2.g4 to deal with variable declarations…

ANTLR Syntax-directed translation

- You can give your own names to symbols in case you have multiple occurrences
- Example:

```
something :
    {int a,b;}
    a=NAME EQUAL b=NAME SEMICOLON
    {System.out.println($a.text + "-" + $b.text);}
    ;
```

ANTLR Syntax-directed translation

- You can create attributes for non-terminal grammar symbols and use them
- Example:

```
something :
        ident SEMICOLON
        {System.out.println("stuff"+$ident.whatever);}
        ;

ident returns [String whatever]:
        NAME
        {$whatever = "somestring"+$NAME.text;}
        ;
```

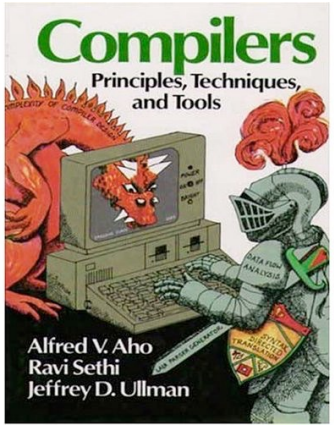
ANTLR Syntax-directed translation

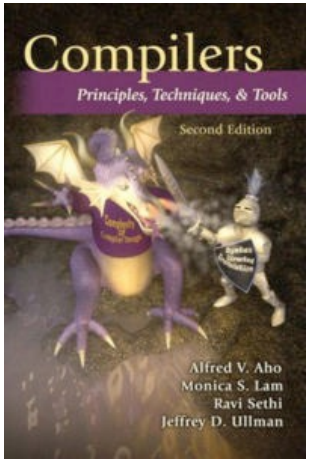
- And with all this we can now implement our compiler
- Our goal: have ANTLR produce x86 assembly code that we can run!
- Let's do it in class right now on my Linux VM…
  - There will be mistakes, questions, hiccups, and confusion… it will take a while
  - But the goal is to learn from this
  - Feel free to suggest things to add to our language!

- Afterward we can look at the MyLanguageV2.g4 file posted in the "A simple ANTLR compiler" reading in the module…

Conclusion

- There is a LOT of depth to the topic of Compilers
- We've only scratched the surface here
- There are well-known books on compilers





- Let's now talk a bit about code optimization in the next module…