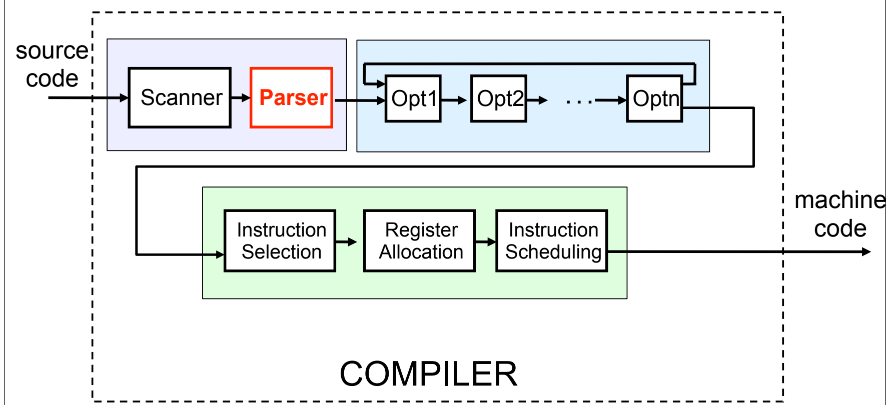
Syntactic Analysis

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The Big Picture Again



Syntactic Analysis

- Lexical Analysis was about ensuring that we extract a set of valid words (i.e., tokens/lexemes) from the source code
- But nothing says that the words make a coherent sentence (i.e., program)
- Example:
  - "if while i == == == 12 + endif abcd"
  - Lexer will produce a stream of tokens: **<TOKEN_IF> <TOKEN_WHILE> <TOKEN_NAME, "i"> <TOKEN_EQUAL> <TOKEN_EQUAL> <TOKEN_EQUAL> <TOKEN_INTEGER,"12"> <TOKEN_PLUS, "+"> <TOKEN_ENDIF> <TOKEN_NAME, "abcd">**
  - This program is lexically correct, but syntactically incorrect

Grammar

- Question: How do we determine that a sentence is syntactically correct?
- Answer: We check against a grammar!
- A grammar consists of rules that determine which sentences are correct
- Example in English:
  - A sentence must have a verb
- Example in C:
  - A "{" must have a matching "}"

Grammar

- Regular expressions are one way we have seen for specifying a set of rules
- Unfortunately they are not powerful enough for describing the syntax of programming languages
- Example:
  - If we have 10 '{' then me must have 10 '}'
  - We can't implement this with regular expressions because they do not have memory!
    - no way of counting and remembering counts
- Therefore we need a more powerful tool
- This tool is called Context-Free Grammars
  - And some additional mechanisms

Context-Free Grammars

- A context-free grammar (CFG) consists of a set of production rules
- Each rule describes how a non-terminal symbol can be "replaced" or "expanded" by a string that consists of non-terminal symbols and terminal symbols
  - Terminal symbols are really tokens
  - Rules are written with syntax like regular expressions
- Rules can then be applied recursively
- Eventually one reaches a string of only terminal symbols, or so one hopes
- This string is syntactically correct according to the grammatical rules!
- Let's see a simple example

CFG Example

- Set of non-terminals: A, B, C (uppercase initial)
- Start non-terminal: S (uppercase initial)
- Set of terminal symbols: a, b, c, d
- Set of production rules:

  S ➔ A | BC
  A ➔ Aa | a
  B ➔ bBCb | b
  C ➔ dCcd | c
- We can now start producing syntactically valid strings by doing derivations
- Example derivations:

  S ➔ BC ➔ bBCbC ➔ bbCbC ➔ bbdCcdbC ➔ bbdccdbC ➔ bbdccdbc
  S ➔ A ➔ Aa ➔ Aaa ➔ Aaaa ➔ aaaa

A Grammar for Expressions

Expr	➔ Expr Op Expr
Expr	➔ Number | Identifier
Identifier	➔ Letter | Letter Identifier
Letter	➔ a-z
Op	➔ "+" | "-" | "*" | "/"
Number	➔ Digit Number | Digit
Digit	➔ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

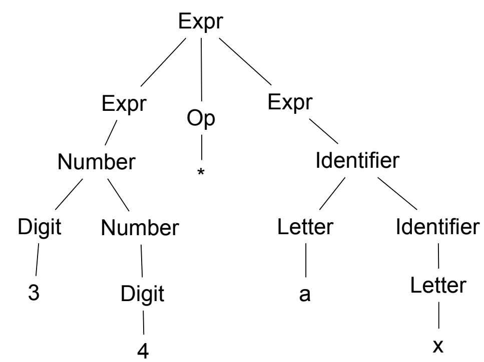
Expr ➔ Expr Op Expr ➔ Number Op Expr ➔
Digit Number Op Expr ➔ 3 Number Op Expr ➔ 34 Op Expr ➔
34 * Expr ➔ 34 * Identifier ➔ 34 * Letter Identifier ➔
34 * a Identifier ➔ 34 * a Letter ➔ 34 * ax

What is Parsing?

- What we just saw is the process of, starting with the start symbol and, through a sequence of rule derivations, obtain a string of terminal symbols
  - We could generate all correct programs (infinite set though)
- Parsing: the other way around
  - Give a string of non-terminals, the process of discovering a sequence of rule derivations that produce this particular string
- When we say we can't parse a string, we mean that we can't find any legal way in which the string can be obtained from the start symbol through derivations
- What we want to build is a parser: a program that takes in a string of tokens (terminal symbols) and discovers a derivation sequence, thus validating that the input is a syntactically correct program

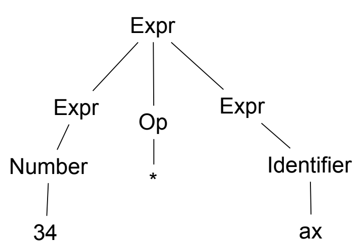
Derivations as Trees

- A convenient and natural way to represent a sequence of derivations is a syntactic tree or parse tree
- Example: Expr ➔ Expr Op Expr ➔ Number Op Expr ➔ Digit Number Op Expr ➔ 3 Number Op Expr ➔ 34 Op Expr ➔ 34 * Expr ➔ 34 * Identifier ➔ 34 * Letter Identifier ➔ 34 * a Identifier ➔ 34 * a Letter ➔ 34 * ax



Derivations as Trees

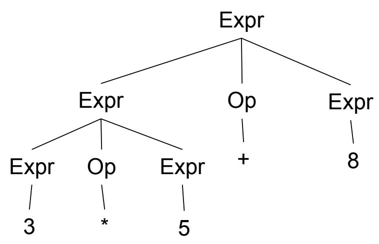
- In the parser, derivations are implemented as trees
- Often, we draw trees without the full derivations
- Example:



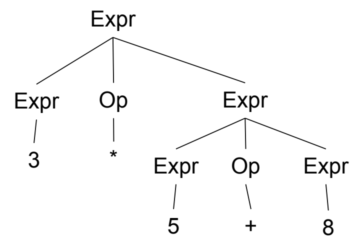
Ambiguity

- We call a grammar ambiguous if a string of terminal symbols can be reached by two different derivation sequences
- In other terms, a string can have more than one parse tree
- It turns out that our expression grammar is ambiguous!
- Let's show that string 3*5+8 has two parse trees

Ambiguity



“left parse-tree”



“right parse-tree”

Problems with Ambiguity

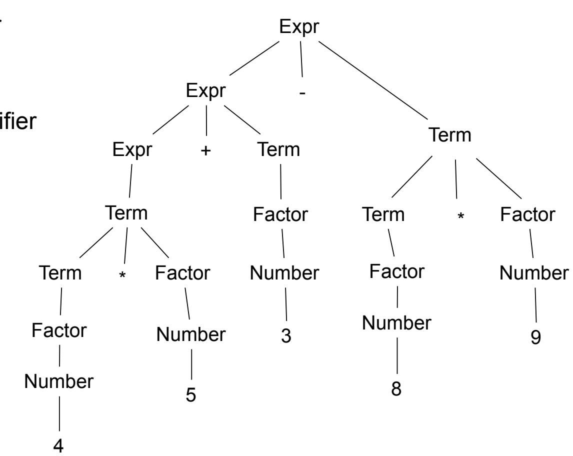
- Problem: syntax impacts meaning
- For our example string, we’d like to see the left tree because we most likely want * to have a higher precedence than +
- We don’t like ambiguity because it makes the parsers difficult to design because we don’t know which parse tree will be discovered when there are multiple possibilities
- So we often want to disambiguate grammars
- It turns out that it is possible to modify grammars to make them non-ambiguous
  - by adding non-terminals
  - by adding/rewriting production rules
- In the case of our expression grammar, we can rewrite the grammar to remove ambiguity and to ensure that parse trees match our notion of operator precedence
  - We get two benefits for the price of one
  - Would work for many operators and many precedence relations

Non-Ambiguous Grammar

Expr → Term | Expr + Term | Expr - Term
Term → Term * Factor
| Term / Factor
| Factor

Factor → Number | Identifier

Example: 4*5+3-8*9

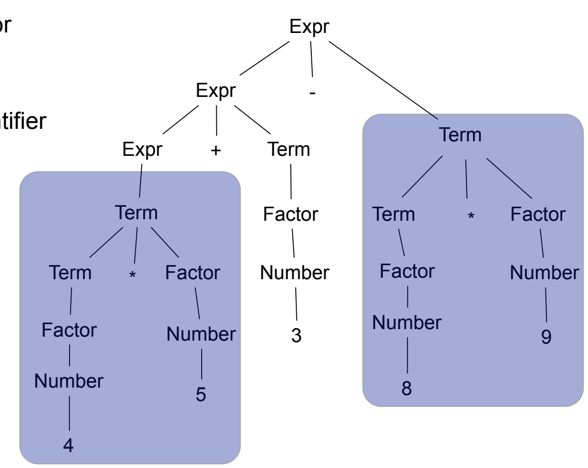


Non-Ambiguous Grammar

Expr → Term | Expr + Term | Expr - Term
Term → Term * Factor
| Term / Factor
| Factor

Factor → Number | Identifier

Example: 4*5+3-8*9



Another Example Grammar

ForStatement → for "(" StmtCommaList ";" ExprCommaList ";" StmtCommaList ")" "{" StmtSemicList "}"

StmtCommaList → ε | Stmt | Stmt "," StmtCommaList

ExprCommaList → ε | Expr | Expr "," ExprCommaList

StmtSemicList → ε | Stmt | Stmt ";" StmtSemicList

Expr → . . .

Stmt → . . .

Full Language Grammar Sketch

Program → VarDeclList FuncDeclList

VarDeclList → ε | VarDecl | VarDecl VarDeclList

VarDecl → Type IdentCommaList ";"

IdentCommaList → Ident | Ident "," IdentCommaList

Type → int | char | float

FuncDeclList → ε | FuncDecl | FuncDecl FuncDeclList

FuncDecl → Type Ident "(" ArgList ")" "{" VarDeclList StmtList "}"

StmtList → ε | Stmt | Stmt StmtList

Stmt → Ident "=" Expr ";" | ForStatement | ...

Expr → ...

Ident → ...

Using * notations (not + here)

Program → VarDeclList FuncDeclList

VarDeclList → VarDecl*

VarDecl → Type IdentCommaList ";"

IdentCommaList → Ident ("," Ident)*

Type → int | char | float

FuncDeclList → FuncDecl*

FuncDecl → Type Ident "(" ArgList ")" "{" VarDeclList StmtList "}"

StmtList → Stmt*

Stmt → Ident "=" Expr ";" | ForStatement | ...

Expr → ...

Ident → ...

Real-world CFGs

- Some sample grammars found on the Web
  - LISP: 7 rules
  - PROLOG: 19 rules
  - Java: 30 rules
  - C: 60 rules
  - Ada: 280 rules
- LISP is particularly easy because
  - No operators, just function calls
  - Therefore no precedence, associativity
- LISP is thus very easy to parse
- In the Java specification the description of operator precedence and associativity takes 25 pages

So What Now?

- We want to write a compiler for a given language
- Lexing
  - We come up with a definition of the tokens embodied in regular expressions
  - We build a lexer using a tool
  - In the previous set of lecture notes, we have used ANTLR to do this
- Parsing
  - We come up with a definition of the syntax embodied in a context-free grammar
  - We build a parser using a tool
  - Let's use ANTLR again for a simple language!

Our Language

- We have all the tokens we've already defined in our lexer:
  - IF, ENDIF
  - PRINT, INT, PLUS, LPAREN, RPAREN
  - EQUAL, NOTEQUAL, ASSIGN, SEMICOLON
  - INTEGER, NAME
- We want a very limited language with
  - integer variable declarations
  - assignments
  - addition (only 2 operands)
  - if (not else, only test for equality)
  - semicolon-terminated statements
  - white-spaces, tabs, carriage returns don't matter
- Let's look at an example program to get a sense of it

Example Program

```
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 b;
```

Let's write/run the grammar

- Root non-terminal: program
- Let us now write the grammar in class together using ANTLR syntax...
  - Using our simple Lexer as a starting point
- A (hopefully similar) grammar is posted on the course Website

Code Generation

- Now we have a parser that will reject syntactically incorrect code, and generate a parse tree for correct code
- The next step toward building a compiler is to generate code
- 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
- There is quite a bit of theory here, but instead we'll just do it by example using the ANTLR syntax
- First let's just review a few basic elements of this syntax

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!");}
     statements*
    {System.out.println("Done!");}
     ;
```

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

- 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
  - A (hopefully) similar version is posted on the course Web site
- There will be mistakes, questions, hiccups, and confusion
- But the goal is that we can all learn from this?
- Off we go....

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