Introduction to Algorithmic Differentiation

Derivative Code Automatically (Part II: Syntax Analysis)

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Contents

Syntax Analysis

Parser Generator `bison`

Syntax analysis aims to transform the given sequence of tokens into an abstract syntax tree (AST) representing a derivation based on the context-free part of the given grammar.

A program for performing syntax analysis is also referred to as parser.

Parsers can be generated automatically, e.g. by the tool `bison`.

`https://www.gnu.org/software/bison/`

Outline

SL^2: Straight-Line Simple Language

Shift-Reduce Parsing

- Resolution of Ambiguity
- `bison`

Compiler Front-End

SL^2: Straight-Line Simple Language I

An SL^2 program is a sequence of statements described by the grammar $G = (V_n, V_t, P, s)$ with nonterminal symbols

$$V_n = \left\{ \begin{array}{ll} s & \text{(sequence of statements)} \\ a & \text{(assignment)} \\ e & \text{(expression)} \end{array} \right\}$$

terminal symbols

$$V_t = \left\{ \begin{array}{rl} V & \text{(program variables)} \\ C & \text{(constants)} \\ F & \text{(unary intrinsic)} \\ L & \text{(linear binary arithmetic operator)} \\ N & \text{(nonlinear binary arithmetic operator)} \\ ;\ )\ (\ = & \text{(remaining single character tokens)} \end{array} \right\}$$

Syntax Analysis

SL^2: Straight-Line Simple Language II

start symbol s, and production rules

$$P = \left\{ \begin{array}{llllll} (P1) & s : a & (P2) & s : as & (P3) & a : V = e; \\ (P4) & e : eLe & (P5) & e : eNe & (P6) & e : F(e) \\ (P7) & e : V & (P8) & e : C & & \end{array} \right\} \quad .$$

Ambiguity due to associativity $a * b * c \overset{?}{=} (a * b) * c \overset{?}{=} a * (b * c)$ and/or operator precedence $a + b * c \overset{?}{=} (a + b) * c \overset{?}{=} a + (b * c)$ needs to be resolved by the parser.

Sample SL^2-Program:

```
x=x*sin(x*y);
y=x/y;
x=cos(x);
```

Outline

SL^2: Straight-Line Simple Language

Shift-Reduce Parsing

- Resolution of Ambiguity
- `bison`

Compiler Front-End

Syntax Analysis

Shift-Reduce Parser

Consider $y = \sin(x * 2)$; i.e. $V = F(VNC)$; Shift symbols onto stack. Reduce top of stack to left-hand side of production rule as soon as possible:

$V = F(V$	$e :: V$
$V = F(eNC$	$e :: C$
$V = F(eNe$	$e :: eNe$
$V = F(e)$	$e :: F(e)$
$V = e;$	$a :: V = e;$
a	$s :: a$
s	

Reductions in reverse order yield right-most derivation.

$$s \rightarrow a \rightarrow V = e; \rightarrow V = F(e); \rightarrow V = F(eNe);$$

$$\rightarrow V = F(eNC); \rightarrow V = F(VNC);$$

Live: (Abstract) Syntax Tree (AST)

Syntax Analysis

Resolution of Ambiguity with `bison`: Associativity

`bison` generates operator precedence shift-reduce parsers with single token *lookahead*.

Example: `x*y*z` $\Rightarrow$ $VNVNV$ for

```
%left N // reduce asap
```

$V[N]$	lookahead N infeasible $\Rightarrow$ reduce
e	no reduction possible $\Rightarrow$ shift
eN	no reduction possible $\Rightarrow$ shift
$eNV[N]$	lookahead N infeasible $\Rightarrow$ reduce
$eNe[N]$	lookahead N feasible $\Rightarrow$ shift or reduce?
	shift-reduce conflict resolved by “%left N” $\Rightarrow$ reduce
e	no reduction possible $\Rightarrow$ shift
eN	no reduction possible $\Rightarrow$ shift
eNV	empty lookahead $\Rightarrow$ reduce
eNe	empty lookahead $\Rightarrow$ reduce
e	

Syntax Analysis

Resolution of Ambiguity with `bison`: Operator Precedence

Consider `x+y*z` ⇒ $VLVNV$ for

```
%left L
%left N // N takes precedence over L
```

and bison default behavior ("reduce asap").

$V[L]$	lookahead L infeasible ⇒ reduce
e	no reduction possible ⇒ shift
eL	no reduction possible ⇒ shift
$eLV[N]$	lookahead N infeasible ⇒ reduce
$eLe[N]$	lookahead N feasible ⇒ shift or reduce ?
	shift-reduce conflict resolved by operator precedence ⇒ shift
$eLeN$	no reduction possible ⇒ shift
$eLeNV$	empty lookahead ⇒ reduce
$eLeNe$	empty lookahead ⇒ reduce
eLe	empty lookahead ⇒ reduce
e	

Syntax Analysis

Parser Generator `bison`: `parser.y`; Part I

```
%token V C F L N

%left L
%left N

%%

s : a
  | a s
  ;
a : V '=' e ';' ;
e : e L e
  | e N e
  | F '(' e ')'
  | V
  | C
  ;

%%
```

Syntax Analysis

Parser Generator `bison`: `parser.y`; Part II

```
#include<stdio.h>

int yyerror(char *msg) { printf("ERROR: %s \n",msg); return -1; }

int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

Syntax Analysis

`bison`: Diagnostics

- `bison -v parser.y` generates `parser.output`

```
Grammar

    0 $accept: s $end
    ...
```

- `bison -g parser.y` generates `parser.dot`

```
digraph "parser.y"
{
    ...
```

- `dot -Tpdf parser.dot >parser.pdf` generates `parser.pdf`
- The parser implements a push-down automaton (DFA + stack).

→ live demo + discussion

Outline

SL^2: Straight-Line Simple Language

Shift-Reduce Parsing

- Resolution of Ambiguity
- `bison`

Compiler Front-End

Source Transformation Front-End

Implementation with `flex` and `bison`

To be provided by the user:

- `flex` input file: `scanner.l`
- `bison` input file: `parser.y`

Build process (makefile):

```
parse : parser.tab.c lex.yy.c
        gcc $^ −lfl −o $@

parser.tab.c : parser.y
        bison −d −o $@ $<

lex.yy.c : scanner.l
        flex $<
```

Source Transformation Front-End

Case Study: AST Printer

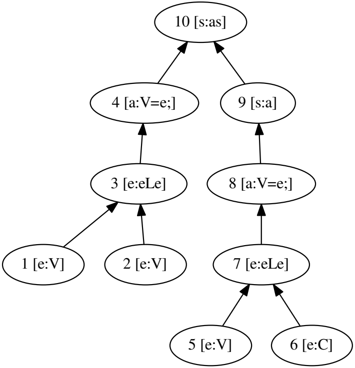
Inspection of

- `scanner.l`
- `parser.y`

and application to

```
t=t+dt;
i=i+1;
```

yields AST on the right.



Construction of intermediate representation in memory facilitates program analysis, transformation and optimization.

Summary

SL^2: Straight-Line Simple Language

Shift-Reduce Parsing

- Resolution of Ambiguity
- `bison`

Compiler Front-End