Introduction to Algorithmic Differentiation

Derivative Code Automatically (Part I: Lexical Analysis)

Uwe Naumann

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

RWTH Aachen

Contents

Outline

Motivation: dcc

The prototype derivative code compiler dcc generates first- and higher-order tangent and adjoint code for a simple subset of C++.

Live:

```
void f(int n, double& x, double* p) {
  double dt=0; double t=0; int i=0;
  dt=1.0/n;
  while (i<n) {
    x=x+dt*p[i]*sin(x*t);
    t=t+dt;
    i=i+1;
  }
}
```

Outline

`f.c`

Scanner (Lexical Analyzer)

Sequence of Tokens

Parser (Syntax Analyzer) $\rightarrow$ `f1.c`

Internal Representation(e.g., parse tree and symbol table)

Control-/Data-Flow Engine (Static Program Analysis)

Annotated Internal Representation

Unparser

`f2.c`

```
void f(int n, double& x, double* p) {
  double dt=0; double t=0; int i=0;
  dt=1.0/n;
  while (i<n) {
    x=x+dt*p[i]*sin(x*t);
    t=t+dt;
    i=i+1;
  }
}
```

```
VOID SYMBOL(INT SYMBOL,
    FLOAT& SYMBOL, FLOAT*
      SYMBOL) {
  FLOAT SYMBOL=CONSTANT;
  FLOAT SYMBOL=CONSTANT;
  INT SYMBOL=CONSTANT;
  SYMBOL=CONSTANT/SYMBOL;
  WHILE (SYMBOL<SYMBOL) {
    SYMBOL=SYMBOL
      +SYMBOL*SYMBOL[SYMBOL]
      *SIN(SYMBOL*SYMBOL);
    SYMBOL=SYMBOL+SYMBOL;
    SYMBOL=SYMBOL+CONSTANT;
  }
}
```

Symbol table stores all SYMBOLs (f,n,x,p,dt,t,i).

The Story in a Nutshell

Syntax Analysis

... based on syntax (also: production) rules, for example, declarations

```
argument: INT SYMBOL
        | FLOAT '&' SYMBOL
        | FLOAT sequence_of_asterixes SYMBOL
...
local_declaration: FLOAT SYMBOL '=' CONSTANT ';'
                 | INT SYMBOL '=' CONSTANT ';'
```

yielding further information in symbol table

name	kind	type	shape
f	1	0	0
n	2	2	1
x	2	1	1
p	2	1	2
dt	2	1	1
t	2	1	1
i	2	2	1

with kinds (subroutine – 1 or variable – 2), types (FLOAT – 1 or INT – 2), and shapes (scalar – 1 or vector – 2) and ..

The Story in a Nutshell

Parse Tree (e.g, subtree)

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

with production rules

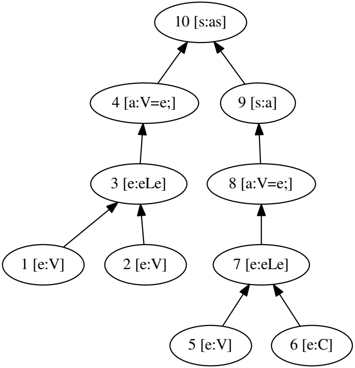
```
sequence_of_statements: statement
    | sequence_of_statements statement

statement: assignment

assignment: memref '=' expression ';'

expression: expression '+' expression
      | memref
      | CONSTANT

memref: SYMBOL
```



E.g, type checking

```
std::string s="42"; int i=s;
```

We rely on syntactically and semantically correct input codes, for example, to be verified by a standard C++ compiler.

E.g, actvity of x,y,z as fixed point after two iterations for

```
while (c) {
  y=y+cos(z);
  z=sin(x);
}
```

and originally varied x and useful y.

- parse tree printer
- unparser
- single assignment code generator
- tangent code generator
- adjoint code generator

and all this in syntax-directed regime (driven by attribute grammar and without explicit generation of parse tree) if possible.

Outline

Alphabets are finite, nonempty sets of symbols (e.g, ASCII characters).

Strings (or words) are finite sequences of symbols from an alphabet Σ (e.g, sequences of ASCII characters). The empty string has zero occurrences of symbols from Σ. It is denoted ϵ.

Languages are all $L \subseteq \Sigma^*$ (e.g, C++)

A Deterministic Finite Automaton (DFA) is a quintuple

$$A = (Q, \Sigma, \delta, q_0, F)$$

where

1. Q is a finite set of states
2. Σ is a finite alphabet (input symbols)
3. δ is a transition function $(q_i, \sigma) \mapsto q_j$ where $\sigma \in \Sigma$ and $q_i, q_j \in Q$
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of final states

Live: v01

A Nondeterministic Finite Automaton (NFA) is a quintuple

$$A = (Q, \Sigma, \delta, q_0, F)$$

where

1. Q is a finite set of states
2. Σ is a finite alphabet (input symbols)
3. δ is a transition function $(q_i, \sigma) \mapsto Q^*$ where $\sigma \in \Sigma \cup \{\epsilon\}$
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of final states

Live: v01

A grammar G is a quadruple

$$G =< V_t, V_n, S, P >$$

where

- V_t is a finite set of terminal symbols, e.g, English words.
- V_n is a finite set of non-terminal symbols, e.g, English sentences.[1]
- $S \in V_n$ is the start symbol, e.g, valid English text.
- P is a finite set of production rules of the form

$$u \rightarrow v \ ,$$

e.g, `sentence` $\rightarrow$ `noun verb '.'` (such as: Corona rocks. ... the beer ...)

[1] $V_t \cap V_n = \varnothing$

- Type 0: Phrase structure grammars
- Type 1: Context sensitive grammars
- Type 2: Context-free grammars. All productions have the form $A \rightarrow v$ where

$$A \in V_n \quad \wedge \quad v \in (V_n \cup V_t)^*$$

- Type 3: Regular grammars. A regular grammar is a left or right linear grammar
  - Left linear grammar

$$A \rightarrow Bt \text{ or } A \rightarrow t \quad \text{where} \quad A, B \in V_n, t \in V_t^*$$

  - Right linear grammar

$$A \rightarrow tB \text{ or } A \rightarrow t \quad \text{where} \quad A, B \in V_n, t \in V_t^*$$

A grammar can generate a string if, starting from the start symbol and successively using the production rules, we can produce that string. This process is known as derivation.

The set of strings that can be derived forms the language generated by the grammar.

Example: Let $G = (V_t, V_n, s, P)$ with $V_t = \{W, O\}$, $V_n = \{a, b, c, d\}$, $s = a$, and production rules $a \rightarrow Wb$, $b \rightarrow Oc$, $b \rightarrow Ob$, $c \rightarrow Wd$, $d \rightarrow \epsilon$.

One derives

$$a \Rightarrow^* WOb \Rightarrow^* WOOOWd \Rightarrow WOOOW$$

as

$$a \Rightarrow Wb \Rightarrow WOb \Rightarrow WOOb \Rightarrow WOOOc \Rightarrow WOOOWd \quad .$$

Outline

Regular Expressions (RE) are $\emptyset$, ϵ, a, $A_1|A_2$, A_1A_2, $A*$, $A+$, (A), where a is a symbol and A_1, A_2, and A are regular expressions.

Examples: `(01)*|(10)*`   `(01)+(10)+`   `stce`

Lexical analysis aims to cluster the sequence of elements from the given alphabet (e.g, ASCII characters) into tokens based on the regular part of the grammar.

A program for performing lexical analysis is also referred to as scanner.

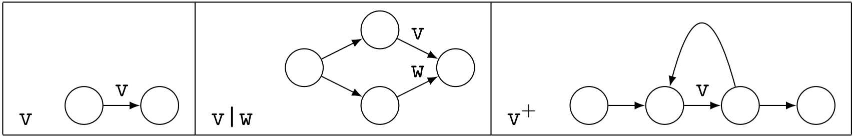
Scanners can be generated automatically, e.g. by the tool `flex`.

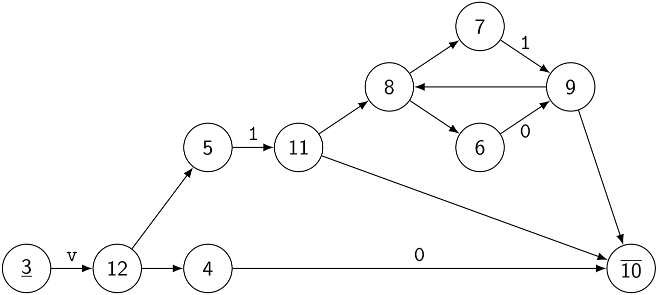
https://www.gnu.org/software/flex/

Lexical Analysis with `flex`

RE → NFA, e.g, v(0|(1(0|1)∗))

Thompson construction, e.g.:





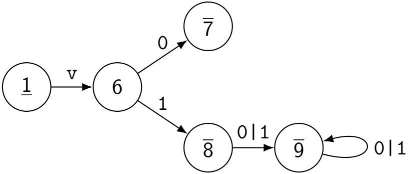
Note: Enumeration consistent with `flex -T` output.

Lexical Analysis with `flex`

NFA → DFA, e.g, v(0|(1(0|1)∗))

Subset construction:

DFA	NFA	v	0	1
$\underline{1}$	$\underline{3}$	$\{12,4,5\}$		
6	$\{12,4,5\}$		$\{10\}$	$\{11,10,8,6,7\}$
$\overline{7}$	$\{\overline{10}\}$			
$\overline{8}$	$\{11,\overline{10},8,6,7\}$		$\{9,10,8,6,7\}$	$\{9,10,8,6,7\}$
$\overline{9}$	$\{9,\overline{10},8,6,7\}$		$\{9,10,8,6,7\}$	$\{9,10,8,6,7\}$



Note: Enumeration consistent with `flex -T` output. Minimal DFA does not require state 9.

flex Input File: scanner.l

```
variable v(0|(1(0|1)*))

%%

{variable} { }
. { return -1; }

%%

int main() { yylex(); return 0; }
```

Lexical Analysis with `flex`

`flex -T` Output (RE)

```
1 (v(0|1(0|1)*))
2 .
3 End Marker
```

... `flex` accepts everything, i.e.

- tokens defined by RE (1)
- newline character as marker of end of string (3)
- remaining single character tokens (2)

`flex -T` Output (NFA)

```
state # 3 118: 12, 0
state # 4 48: 10, 0
state # 5 49: 11, 0
state # 6 48: 9, 0
state # 7 49: 9, 0
state # 8 257: 6, 7
state # 9 257: 8, 10
state # 10 257: 0, 0 [1]
state # 11 257: 8, 10
state # 12 257: 4, 5
```

[1] refers to RE (v(0|1(0|1)*)); ASCII codes 48,49,118; Empty word: 257

Live: NFA

Note: Restriction to essential contents of `flex -T` output.

flex -T Output (DFA)


```
state # 1:
          ...
          5 6 // 5 -> v
...
state # 6:
          3 7 // 3 -> 0
          4 8 // 4 -> 1
state # 7:
state # 8:
          3 9
          4 9
state # 9:
          3 9
          4 9
...
state # 7 accepts: [1]
...
```

Live: NFA

Note: Restriction to essential contents of `flex -T` output.

Lexical Analysis with flex

Error Handling

... is up to the user, e.g.

```
%{
#include<stdio.h>
%}

regex v(0|1(0|1)*)

%%

{regex} { printf("%s\n",yytext); }
. { printf("ERROR: %c\n",yytext[0]); return -1; }

%%

int main() { yylex(); return 0; }
```

Live: Demo

Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

```
%{
#include<stdio.h>
%}

d double

%%

{d} { printf("dco::ga1s<double>::type"); }
. { printf("%c",yytext[0]); }

%%

int main() { yylex(); return 0; }
```

Live: Demo

Summary

Motivation

The Story in a Nutshell

Terminology

- Deterministic Finite Automaton (DFA)
- Nondeterministic Finite Automaton (NFA)
- Grammar
- Derivation

Lexical Analysis

- Regular Expressions (RE)
- RE $\rightarrow$ NFA
- NFA $\rightarrow$ DFA
- `flex`