Compiler Construction Supervision 1

Matthew Johnson
Trinity Hall
mjj29@cam.ac.uk

February 26, 2003

1 y2001p3q3

1.1 Regular Grammar

$$
\begin{array}{rcl}
F & \rightarrow & 0 \\
F & \rightarrow & 1 \\
F & \rightarrow & 0D \\
F & \rightarrow & 1D \\
F & \rightarrow & 0B \\
F & \rightarrow & 1B \\
B & \rightarrow & . \\
D & \rightarrow & . \\
D & \rightarrow & .N \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
N & \rightarrow & 0N \\
N & \rightarrow & 1N \\
N & \rightarrow & eP \\
P & \rightarrow & 0 \\
P & \rightarrow & 1 \\
P & \rightarrow & 0P \\
P & \rightarrow & 1P
\end{array}
$$

1.2 Non-regular Grammar

$$\begin{aligned} F &\to M|MeN \\ M &\to N|N.|N.N \\ N &\to D|ND \\ D &\to 0|1 \end{aligned}$$

1.3

T can generate adc, whereas S cannot, but S can generate aad or dcc, which T cannot.

1.4 Strings from Grammers

Strings which can be generated by regular grammers are easily recognisably by an automated process, since they correspond directly to a Finite State Machine. This makes them easy to parse, and build a tree of the syntactic elements of the grammer for later conversion to code. This module would be part of the Syntax Analysis stage of the compiler, and would generally call the lexical analyser to get tokens as it parses. The result would be a tree of the parsed program, built up from the grammer, and representing the structure of the program. (give examples of functions in lexer)

2 y2000p4q4

2.1 Left/Right/Follow Sets

The Left set is the set of all things which a Non-terminal can appear on the left of. For example, in this case Left(S) would be E <eof>. The Right set is the set of all things which can have a Non-terminal on the right of them, and the Follow set is the set of all things that can immediately follow a Non-terminal.

Follow(Y) contains: for $F \to .....YX....$: X, Left(X) and for $G \to .....Y$ Follow(G).

For the given grammer. the full table is as follows:

NT	Left(NT)	Right(NT)	Follow(NT)
S	{"E <eof>"}	{}	{}
E	{"T","T + E"}	{"S","E"}	{"<eof>"}
T	{"x"}	{"E"}	{"+","<eof>"}

3 y1999p6q6

3.1 context free grammar

$$\begin{array}{ccl} \text{S} & \rightarrow & E \\ \text{E} & \rightarrow & P|B1|B2|E*E \\ \text{P} & \rightarrow & a|b|c \\ \text{B1} & \rightarrow & (+E) \\ \text{B2} & \rightarrow & [+E] \end{array}$$

3.2 Floating Point Grammar

target: $[+-]dd*[.d*][e[+-]dd*]$, where we use $s = +, -$, $d = 0-9$ and we divide the regexp up into Signed (S), Unsigned (S), Digits (D), Fraction (F) and Exponent (E).

$$\begin{aligned} S &\rightarrow sU|U \\ U &\rightarrow D|DF|DE|DFE \\ D &\rightarrow d|dD \\ F &\rightarrow .|.D \\ E &\rightarrow eD|esD \end{aligned}$$

4 1998p4q4

4.1 Recursive Descent Parser

(See drawn diagram)

```
void descendSentence()
{
   lex();
   descendExpr();
   if (token != EOF) error("expected eof");
}

void descendExpr()
{
   descendTerm();
   for (;;) switch(token) {
      case '+': {
         lex();
```

```
            descendTerm();
            continue;
         }
         case '-': {
            lex();
            descendTerm();
            continue;
         }
         default: return;
      }
   }

   void descendTerm()
   {
      descendPrimary();
      for (;;) switch (token) {
         case '^': {
            lex();
            descendPrimary();
            continue;
         }
         default: return;
      }
   }

   void descendPrimary()
   {
      switch (token) {
         case '(': {
            lex();
            descendExpr();
            if (token != ')') error("'(' expected");
            lex();
            return;
         }
         case 'n': lex(); return;
         default: error("Unexpected Token" + token);
      }
   }
```

4.2 Table-driven parser

Rules

r1	S	$\rightarrow$	E eof
r2	E	$\rightarrow$	E + T
r3	E	$\rightarrow$	E - T
r4	E	$\rightarrow$	T
r5	T	$\rightarrow$	P ** T
r6	T	$\rightarrow$	P
r7	P	$\rightarrow$	(E)
r8	P	$\rightarrow$	n

CFSM reductions:

```
1: {
      S -> .E eof    E => 2
      E -> .T        T => 5
      E -> .E + T    E => 2
      E -> .E - T    E => 2
      T -> .P        P => 6
      T -> .P ^ T    P => 6
      P -> .n        n => 11
      P -> .( E )    ( => 12
   }
2: {
      S -> E .eof    eof => accept // r1
      E -> E .+ T    + => 3   // do these both
      E -> E .- T    - => 3   // go to the same?
   }
3: {
      E -> E + .T    T => 4   // ditto, probably
      E -> E - .T    T => 4   // not, I think.
      T -> .P ** T   P => 10
      T -> .P        P => 10
      P -> .n        n => 11
      P -> .( E )    ( => 12
   }

// duplicate the above & below
// rules for + & - ??

4: {
      E -> E + T.    r2 => $
      E -> E - T.    r3 => $
   }
5: {  E -> T.        r4 => $  }
```

```
6: {
      T -> P.         r6 => $
      T -> P .** T    ** => 8
   }
8: {
      T -> P ** .T    T => 9
      T -> .P ** T    P => 6
      T -> .P         P => 6
      P -> .n         n => 11
      P -> .( E )     ( => 12
   }
9: {  T -> P ** T.    r5 => $  }
11:{  P -> n.         r8 => $  }
12:{
      P -> ( .E )     E => 13
      E -> T          T => 5
      // because we are at an E
      // also include all of 3
      // 3 == "what an E can be"
      E -> .E + T     T => 13
      E -> .E - T     T => 13
      T -> .P ** T    P => 10
      T -> .P         P => 10
      P -> .n         n => 11
      P -> .( E )     ( => 12
   }
13:{
      E -> E .+ T     - => 3   //3a
      E -> E .- T     + => 3   //3b
      P -> ( E .)     ) => 14
   }
14:{  P -> ( E ).     r7 => $  }
```

(See drawn diagram)
State table:

state	eof	(	n	)	+	-	**	E	T	P
S1	-	S12	S11	-	-	-	-	S2	S5	S6
S2	acc	-	-	-	S3a	S3b	-	-	-	-
S3a	-	S12	S11	-	-	-	-	-	S4a	S6
S3b	-	S12	S11	-	-	-	-	-	S4b	S6
S4a	r2	-	-	r2	r2	r2	-	-	-	-
S4b	r3	-	-	r3	r3	r3	-	-	-	-
S5	r4	-	-	r4	r4	r4	-	-	-	-
S6	r6	-	-	r6	r6	r6	S8	-	-	-
S8	-	12	11	-	-	-	-	-	S9	S6
S9	r5	-	-	r5	r5	r5	-	-	-	-
S11	r8	-	-	r8	r8	r8	r8	-	-	-
S12	-	S12	S11	-	-	-	-	-	S5	S6
S13	-	-	-	S14	-	-	-	-	-	-
S14	r7	-	-	r7	r7	r7	r7	-	-	-

Note: what are we meant to do for this in an exam - it took me a couple of hours to do this, with notes!

5 notes

5.1 grammars

The term $E \to E * E$ is ambiguous. To make this unambiguous, convert to $E * F|F$, and move the rest of $E \to$ to F

so: $E \to (+E)|[-E]|(E)|[E]|E * E|a|b|c$ goes to $E \to F|E * F$ and $F \to (+F)|[-F]|(F)|[F]|a|b|c$

5.2 Left/Right/Follow

$$
\begin{array}{rcl}
F & \to & M|MeN \\
M & \to & N|N.|N.N \\
N & \to & D|ND \\
D & \to & 0|1
\end{array}
$$

The Left() set of a Non-Terminal is any terminal or non-terminal that can start a production of that Non Terminal.

eg $Left(F) = \{M, N, D, 0, 1\}$

The Right() set of a Non-Terminal is any terminal or non-terminal that can end a production of that Non Terminal.

e.g. $Right(F) = \{N, D, 0, 1\}$

The Follow() set of a symbol is any symbol that can follow that symbol.
e.g. $Follow(M) = \{e\}$

5.3 CFSM -¿ action/goto

For reduction transforms, set `action[state,S]` to the reduce transform for all `S` in the Follow() set of the Non-Terminal that started that production. (eg $E \rightarrow T$. under $r3$ would set all of Follow(T) to $r3$.