Introduction to Syntax Directed Translation and Top-Down Parsers
**Attributes and Semantic Rules**

Let’s associate attributes with grammar symbols, and semantic rules with productions. This gives us a syntax directed definition. (Note simpler grammar)

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E₁ + T</td>
<td>E.v := E₁.v + T.v</td>
</tr>
<tr>
<td></td>
<td>E.v := T.v</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.v := T₁.v * F.v</td>
</tr>
<tr>
<td></td>
<td>T.v := F.v</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.v := E.v</td>
</tr>
</tbody>
</table>
| F → num     | F.v := num.v    | ("value" of token)
Annotated ("Decorated") Parse Tree After Executing Semantic Rules

\[ E \rightarrow E_1 + T \]
\[ \mid T \]
\[ T \rightarrow T_1 * F \]
\[ \mid F \]
\[ F \rightarrow (E) \]
\[ \mid \text{num} \]

E.v := E_1.v + T.v
E.v := T.v
T.v := T_1.v * F.v
T.v := F.v
F.v := E.v
F.v := num.v

Input string: 3 * 4 + 5 * 2
Infix and Postfix Expressions

- Infix: $3 \times 4$
- Postfix: $3 \ 4 \ \times$

- Infix: $3 \times 4 + 5 \times 2$
- Postfix: $3 \ 4 \ \times \ 5 \ 2 \ \times \ +$

- Let’s build a translator from infix to postfix
Translation as an Attribute

An attribute could be anything, including a translation. Let attribute “.t” be a character string, and “||” be the string concatenation operator for strings.

**Grammar**

- $E \rightarrow E_1 + T$
- $E \rightarrow T$
- $T \rightarrow T_1 \ast F$
- $T \rightarrow F$
- $F \rightarrow (E)$
- $F \rightarrow \text{num}$

**Semantic Rules**

- $E.t := E_1.t || T.t || '+'$
- $E.t := T.t$
- $T.t := T_1.t || F.t || '*'$
- $T.t := F.t$
- $F.t := E.t$
- $F.t := \text{num.t}$
Annotated Parse Tree After Executing Semantic Rules

\[ E \rightarrow E_1 + T \quad \text{E.t} := E_1.t || T.t || '+' \]
\[ \mid T \quad \text{E.t} := T.t \]
\[ T \rightarrow T_1 * F \quad \text{T.t} := T_1.t || F.t || '*' \]
\[ \mid F \quad \text{T.t} := F.t \]
\[ F \rightarrow (E) \quad \text{F.t} := E.t \]
\[ \mid \text{num} \quad \text{F.t} := \text{num.t} \]

Input string: \( 3 * 4 + 5 * 2 \)
Translation Schemes

• What we have seen so far is called a “Syntax-directed definition”
  - Which is simply a specification of the translation to be performed
  - Without necessarily addressing how this might be accomplished in any real implementation
    • Order of evaluation of rules is not specified, for example
• If we can develop a plan for how to implement the translation, in the form of particular actions to be performed, and the order in which they should be performed, we will have something called a “Syntax-directed translation Scheme”
Translation Schemes

- If we embed program fragments called \textit{semantic actions} \textit{within} the RHS’s of productions, we will have a \textit{translation scheme}.
- This explicitly shows the order of evaluation of the actions.
  - (assuming we can imply the order in which the parse tree will be covered)
- Here is a scheme which \textit{concatenates} simply by writing \textit{(appending)} to a file.
- \textbf{Implied Traversal order:}
- \textit{We must “visit” child nodes before we can visit} an interior node.
- \textit{..and execute the action when we “visit”}

\begin{verbatim}
E → E + T { print ‘+’ }
  | T
T → T * F { print ‘*’ }
  | F
F → ( E )
  | num { print num }  
\end{verbatim}
Translation Scheme in Action

E → E + T { print ‘+’ }
| T
T → T * F { print ‘*’ }
| F
F → ( E )
| num {print num. t }

Input string: 3 * 4 + 5 * 2

Output string: 3 4 * 5 2 * + 9
Depth First Traversal

- The translation scheme depends on actions happening is a certain order.
- It must be bottom-up, left-to-right
- A guaranteed way to do this is to do a depth-first traversal of the parse tree

```plaintext
procedure visit (node n) {
    for each child m of n, from left to right {
        visit (m);
    }
    evaluate semantic rules at node n;
}
```
Depth First Traversal

procedure visit (node n) {
    for each child m of n, left to right {
        visit (m);
    }
}
Our Translation Scheme Re-Visited

E → E + T { print '+' }
  | T
T → T * F { print '*' }
  | F
F → ( E )
  | num {print num.t }

If we could develop a parsing scheme which "visits" the parse tree nodes in this order, we can "execute" our translation scheme while we parse!

Output string:  3  4  *  5  2  *  + 
• “Formal” Definitions of a Parser
  – A parser is some system capable of constructing
    the derivation of any sentence in some language
    \( L(G) \) based on a grammar \( G \).
    • which talks about a derivation
  – A parser for a grammar \( G \) is a program that reads a
    string \( w \) and produces either a parse tree for \( w \) (if
    \( w \) is a sentence of \( G \)) or an error message.
    • which is about building a parse tree
Two Major Classes of Parser

Top-Down Parser: a parser that builds a parse tree starting at the root (the start symbol), building downwards to the leaves (tokens).

Bottom-Up Parser: a parser that starts at the leaf nodes (bottom), and builds upwards towards the root.

- Has to match up strings of symbols which match the RHS of productions, and reduce them, replacing them with the non-terminal on the LHS.
- Sometimes called a shift-reduce parser.
Top-Down Parser Construction

• Top-down parsers, especially hand-written ones, are easier to understand and visualize the first time around.

• So we’re going to “invent” top-down parsing looking at progressively more interesting grammars.
Parsing a Very Simple Grammar

Grammar G1: \[ S \rightarrow \text{if ( true ) goto label} \]

We need a lexical analyser `getoken()` which identifies one token at a time and returns a token code in the form of a manifest (or enumerated) constant (to use vanilla C terminology). Something like:

```c
#define IF 256
#define TRUE 257
...etc....
```

For single-character tokens like the left parenthesis, we can use the ascii code for the character as the code.
G1 Parser (continued)

- We’re describing algorithms, and we’ll stick to procedural code, mostly in “C”
- Here is a useful procedure which will simplify the programming of a parser. Note the use of a “look-ahead” token in the variable tok.

```c
int tok; /* Yes, a global variable! */
match(int t) {
    if (tok == t) tok = getoken();
    else error("Syntax error");
}
```
The Actual (G1) Parser

The Grammar: \[ S \rightarrow \text{if ( true ) goto label} \]

\[
\begin{align*}
\text{main()} & \{ \\
& \quad \text{tok = getoken(); /* read first lookahead */} \\
& \quad S(); \\
& \quad \text{printf("Parse complete!");} \\
\} \\
\text{S()} & \{ \\
& \quad \text{match(IF); match(' '); match(TRUE);} \\
& \quad \text{match(' '); match(GOTO); match(LABEL);} \\
\}
\]
Dealing with Non-Terminal Symbols

Grammar G2:  
\[
\begin{align*}
S & \rightarrow \text{if ( E ) goto label} \\
E & \rightarrow \text{id > id}
\end{align*}
\]

Approach: simply write a new parsing procedure for each non-terminal symbol in the grammar, and instead of “matching”, if we encounter a non-terminal in some RHS we simply call the parsing procedure for that non-terminal
**G2 Parser**

**G2:**

\[
\begin{align*}
S & \rightarrow \text{if ( E ) goto label} \\
E & \rightarrow \text{id > id}
\end{align*}
\]

**Parser:**

```c
main() { /* same as before */ } 
S() { 
    match(IF); match('(');
    E();
    match(')'); match(GOTO); match(LABEL);
} 
E() { 
    match(ID); match('>'); match(ID);
} 
```
Multiple Productions

A more realistic grammar would generate more than one sentence (!), and require that the parser actually make choices.

For example:

\[ G3: \quad S \rightarrow \text{if } (E) S \\
\quad \quad \quad | \quad \text{goto label} \\
\quad \quad \quad | \quad \text{id = id} \\
\quad E \rightarrow \text{id > id} \]
Multiple Productions

• If there is more than one production for some non-terminal, the parser must decide which one to “apply”

• We will use the look-ahead symbol to make this decision
  - comparing the look-ahead with the first symbol on the LHS of each production
  - but only works if the first symbol is a terminal!
G3 Parser

G3:  

S → if ( E ) S  
     | goto label  
     | id = id  
E → id > id

• main( ) and e( ) have not changed from the previous version
• Note the recursion in S which supports nested if’s
• This is a recursive descent parser

S( ) {  
    if (tok == IF) {
        match(IF); match( '(' );
        E( );
        match( ')' );
        S( );
    }
    else if (tok == GOTO) {
        match(GOTO); match(LABEL);
    }
    else if (tok == ID) {
        match(ID); match( '=' ); match(ID);
    }
    else error("Syntax error in rule S");
}
But Not All Productions Start with Tokens!

How about grammar G4?:

\[
S \rightarrow \ \text{IF\_ST} \\
\ | \ \text{LOOP} \\
\ | \ \text{ASSIGN} \\
\text{IF\_ST} \rightarrow \ \text{if} \ (E) \ S \\
\text{LOOP} \rightarrow \ \text{while} \ (E) \ \text{do} \ S \\
\ | \ \text{do} \ S \ \text{while} \ (E) \\
\text{ASSIGN} \rightarrow \ \text{id} = \ \text{id} \\
E \rightarrow \ \text{id} > \ \text{id}
\]

How will we decide which productions of S to choose in our parser?
When Productions Start With a Non-terminal

| S   | → | IF_ST                  |
|     |   |                       | | • We need to know something about what strings a RHS can derive |
|     |   | |                 | | • Let $A \rightarrow \alpha$ be a production |
|     |   |                       | | • Define $\text{FIRST} (\alpha)$ to be the set of terminals which could be the first token in strings derived from $\alpha$ |
|     |   | |                 | | |
|     |   | |                 | | |
|     |   |                       | | |
| IF_ST | → | if ( E ) S |
| LOOP | → | while ( E ) do S |
|     |   | |   | | |
|     |   |                       | | |
| ASSIGN | → | id = id |
| E   | → | id > id |

• We need to know something about what strings a RHS can derive
• Let $A \rightarrow \alpha$ be a production
• Define $\text{FIRST} (\alpha)$ to be the set of terminals which could be the first token in strings derived from $\alpha$
Example of FIRST( )

A fragment of G4:

\[
\begin{align*}
S & \rightarrow \text{LOOP} \\
\text{LOOP} & \rightarrow \text{while ( E ) do S} \\
       & \mid \text{do S while ( E )}
\end{align*}
\]

What is FIRST (LOOP)?

FIRST (LOOP) = \{ while, do \}

- (the set consisting of the token \text{while} and the token \text{do} )

So select production \( A \rightarrow \alpha \) if the look ahead token is in \text{First} (\( \alpha \))
Choosing the productions of S

We need the FIRST sets for the productions of S

\[ S \rightarrow IF\_ST \]
\[ \mid LOOP \]
\[ \mid ASSIGN \]
\[ IF\_ST \rightarrow \text{if } (E) S \]
\[ LOOP \rightarrow \text{while } (E) \text{ do } S \]
\[ \mid \text{do } S \text{ while } (E) \]
\[ ASSIGN \rightarrow \text{id } = \text{id} \]
\[ E \rightarrow \text{id } > \text{id} \]

FIRST(IF\_ST)={ if }
FIRST(LOOP)={ while, do }
FIRST(ASSIGN)={ id }
Parsing Procedure for S

\[ S() \{ \]
  \[
  \text{if (tok == IF) IF\_ST( );} \\
  \text{else if ((tok == WHILE) || (tok == DO)) LOOP( );} \\
  \text{else if (tok == ID) ASSIGN( );} \\
  \text{else error("Syntax error in rule S");} \\
  \]
\}
Footnotes and Gotcha's

Remember:

\[ S \rightarrow \text{LOOP} \]
\[ \text{LOOP} \rightarrow \text{while ( E ) do S} \]
\[ \quad \mid \text{do S while ( E )} \]

While computing FIRST(LOOP) we may find a production for LOOP that starts with a non-terminal: (must go deeper into grammar..)

..and

If we have two productions \( A \rightarrow \alpha \) and \( A \rightarrow \beta \) then FIRST(\( \alpha \)) and FIRST(\( \beta \)) must be disjoint
Parse Tree Traversal in Recursive Descent Parsers

- Our parser did not produce a parse tree
- ..but it could have, because it constructed a left-most derivation, which is equivalent
- In what order does the parser traverse the conceptual tree?

- Define “visit” to mean:
- We have just called match( ) to consume a leaf node, or:
- We have just called the parsing procedure for a non-terminal (an interior node) and that procedure has just finished matching one of its productions
Parse Tree Traversal

\[ S \rightarrow IF\_ST \]
| \rightarrow LOOP
| \rightarrow ASSIGN
\[ IF\_ST \rightarrow if \ (E) \ S \]
\[ LOOP \rightarrow while \ (E) \ do \ S \]
| \rightarrow do \ S \ while \ (E) \]
\[ ASSIGN \rightarrow id = id \]
\[ E \rightarrow id > id \]

Let's trace:
while \((x > y)\) do \(y = x\)
Recursive Descent is Depth-First!

- This was depth-first, as we defined earlier
- Therefore we could perform a translation scheme using such a parser
- We just insert the actions (code fragments) in the parsing routines, at the points which correspond to “visit”
- Let’s try it!
First Attempt at a Translator

We need to compute FIRST(E) and FIRST(T)

FIRST(E) = { "]", num } 
FIRST(T) = { "(", num }

Whoops!

(Not only do we have no way to decide between the first two E productions, we can’t even distinguish them from the “T one.)

But it gets worse:
if E( ) chooses production E+T, the first thing that happens is it calls E( ) recursively!
Left-Recursion and Top-Down Parsers

• If the first thing a parsing procedure does is call itself recursively without first consuming any input
  - It will make the same decision the next time because the "state" has not changed
  - and call itself again... and again... and again.........

• Bad News:

• We cannot use a recursive descent parser (or any other top-down parser) for a grammar that contains left-recursive productions

• So we need a method to transform grammars containing left-recursion into grammars without left-recursion
Elimination of Left Recursion

• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha \mid \beta \]

• What language does this generate?

• \( S \) generates all strings starting with a \( \beta \) and followed by any number of \( \alpha \)'s

• Can rewrite using right-recursion
  \[ S \rightarrow \beta \ S' \]
  \[ S' \rightarrow \alpha \ S' \mid \epsilon \]
Elimination of Left Recursion

• In general
  
  \[ S \rightarrow S \alpha_1 | ... | S \alpha_n | \beta_1 | ... | \beta_m \]

• All strings derived from \( S \) start with one of \( \beta_1, \ldots, \beta_m \) and continue with several instances of any of the \( \alpha_1, \ldots, \alpha_n \)

• Rewrite as
  
  \[ S \rightarrow \beta_1 S' | ... | \beta_m S' \]
  
  \[ S' \rightarrow \alpha_1 S' | ... | \alpha_n S' | \epsilon \]
General Left Recursion

• The grammar

\[ S \rightarrow A \alpha \mid \delta \]
\[ A \rightarrow S \beta \]

is also left-recursive because

\[ S \Rightarrow S \beta \alpha \]

• This left-recursion can also be eliminated

• See book, pp. 157-162 for general algorithm
Second Attempt at Translator

We will treat the actions just like other grammar symbols:

i.e. "{ foo=bar }" counts as one symbol
Writing a Translator (finally..)

\[
E \rightarrow T \ R
\]

\[
R \rightarrow + \ T \ { \{ \ \text{print '+'} \ \} \ } \ R
| - \ T \ { \{ \ \text{print '-'} \ \} \ } \ R
| \epsilon
\]

\[
T \rightarrow F \ Q
\]

\[
Q \rightarrow * \ F \ { \{ \ \text{print '*'} \ \} \ } \ Q
| / \ F \ { \{ \ \text{print '/'} \ \} \ } \ Q
| \epsilon
\]

\[
F \rightarrow ( \ E )
| \ \text{num} \ { \{ \ \text{print num} \ \} \ }
\]

\[
E( ) \{ \ T() ; \ R() ; \}
\]

\[
R() \{
\quad \text{if (tok=='+')} \{ \\
\quad \quad \text{match('+'); T();} \\
\quad \quad \text{printf(" + "); R();}
\}
\quad \text{else if (tok=='-')} \{ \\
\quad \quad \text{match('-'); T();} \\
\quad \quad \text{printf(" - "); R();}
\}
\quad \text{else ; /* accept empty string */}
\}
\]
Notes

• Note how the semantic action has been carried into the middle of production RHS’s and code inserted at the appropriate point in the parsing routine

• Productions for T and Q are almost identical in form:
The Rest of the Parser/Translator

T( ) {  F( ); Q( ); }  
Q( ) {  
    if (tok=='*') {  
        match('*'); F( );  
        printf(" * "); Q( );  
    }  
    else if (tok=='/') {  
        match('/'); F( );  
        printf(" / "); Q( );  
    }  
    else; /* accept empty string */  
}  

Question: Why did I call printf before calling match(NUM)?

F( ) {  
    if (tok == NUM) {  
        printf(" %d ", tokenval);  
        match(NUM);  
    }  
    else if (tok == '(') {  
        match('('); E( );  
        match(')');  
    }  
    else error("syntax error");  
}
We Need a Lexical Analyser

- We've already seen how straightforward it is to write a "lexer" for a simple grammar such as this.

- Tokens are:
  num  (  )  +  -  *  /  

Here's the header stuff:

```c
#include <stdio.h>
#include <ctype.h>
#include "globals.h"

#define NONE -1    /* used for tokenval */
#define NUM  256
#define ID   257 /* not needed yet */
#define DONE 258

int tok= NONE;
int lineno = 1;
int tokenval = NONE;
```
A Suitable Handwritten Lexer

```c
int getoken() {
    int t;
    while(1) {
        t = getchar();
        if (t==' ' || t=='	') ;
        /*strip white space*/
        else if (t == '
') ++lineno;
        else if (isdigit(t)) { /* num */
            tokenval = t - '0';
            t = getchar();
            while (isdigit(t)) {
                tokenval = tokenval*10 + t-'0';
                t = getchar();
            }
            ungetc(t, stdin);
            return NUM;
        }
        else if (t == EOF) return DONE;
        else {
            tokenval = NONE;
            return t; /* + - ( ) etc. */
        }
    }
}
```

We should really enumerate all the legal one-char tokens and report illegal characters, but if we don’t the parser will end up reporting the error anyway...

Note the use of one-character look-ahead in lexer using a trick from C library
A Second Translator

For a second example, let’s generate “code” for a stack machine with the following instructions:

- push $x$: push contents of word at address $x$
- pushcon $num$: push integer constant $num$
- add: pop 2 words, add them, push sum
- subtract: pop 2 words, subtract, push result
- multiply: pop 2 words, multiply, push product
- divide: pop 2 words, divide, push quotient
Stack Machine Example

\[ x + 22 \]

would translate to:

\begin{verbatim}
push x
pushcon 22
add
\end{verbatim}

(Let’s keep it simple and assume there are only 26 variables a..z: I.e. variable names are single letters)
Translation Scheme for Stack Machine
(Note introduction of variables)

E →  E + T { emit “add”}
    |   E - T { emit “subtract”}
    |   T

T →  T * F { emit “multiply”}
    |   T / F { emit “divide”}
    |   F

F →  id { emit “push ID”}
    |   num { emit “pushcon NUM”}
    |   (   E   )
Remove Left Recursion

\[
\begin{align*}
E & \rightarrow \ T \ R \\
R & \rightarrow \ + \ T \ {\{\text{emit "add"}\}} \ R \\
& \mid - \ T \ {\{\text{emit "subtract"}\}} \ R \\
& \mid \epsilon \\
T & \rightarrow \ F \ Q \\
Q & \rightarrow \ * \ F \ {\{\text{emit "multiply"}\}} \ Q \\
& \mid / \ F \ {\{\text{emit "divide"}\}} \ Q \\
& \mid \epsilon \\
F & \rightarrow \ \text{id} \ {\{\text{emit "push ID"}\}} \\
& \mid \ \text{num} \ {\{\text{emit "pushcon NUM"}\}} \\
& \mid \ ( \ E \ )
\end{align*}
\]
# define NONE -1
# define NUM 256
# define ID 257
# define DONE 258

int getoken()
{
    int t;
    while(1) {
        t = getchar();
        if (t==' ' || t=='\t') /*strip white space*/
            if (t == '\n') ++lineno;
        else if (isdigit(t)) {
            tokenval = t - '0';
            t = getchar();
            while (isdigit(t)) {
                tokenval = tokenval*10 + t-'0';
                t = getchar();
            }
            ungetc(t, stdin); return NUM;
        }
        else if (isalpha(t)) { /* identifier */
            tokenval = tolower(t);
            return ID;
        }
        else if (t == EOF) return DONE;
        else {
            tokenval = NONE;
            return t; /* + - ( ) etc. */
        }
    }
}
Add Code to Generate Assembly Code

/* Functions to simplify printing assembly instructions and report errors */
/* Each prints a single line, hiding formatting details from parser */
void emit(char * x) { /* 0-address instruction (opcode only) */
    printf("\t%s\n",x);
}
void emit1n(char *x, int y) { /* 1-address instruction, numeric parameter */
    printf("\t%s %d\n",x,y);
}
void emit1c(char *x, char y) { /* 1-address instruction, char parameter */
    printf("\t%s %c\n",(x),(y));
}
error(m) char *m; {
    fprintf(stderr,"line %d: %s\n",lineno,m);
    exit(1);
}

Note that these are simple enough that they could actually be written as C "macros
Parser/Translator

E() { T(); R(); }

R() {
    if (tok == '+') {
        match('+'); T();
        emit("add"); R();
    } else if (tok == '-') {
        match('-'); T();
        emit("subtract"); R();
    } else; /* "absorb" empty string */
}

T() { F(); Q(); }

Q() {
    if (tok == '*') {
        match('*'); F();
        emit("multiply"); Q();
    } else if (tok == '/') {
        match('/'); F();
        emit("divide"); Q();
    } else; /* "absorb" empty string */
}
Completion of Translator, and Demo

F(){
    if (tok == ID) {
        emit1c("push",tokenval); match(ID);
    }
    else if (tok == NUM) {
        emit1n("pushcon", tokenval);
        match(NUM);
    }
    else if (tok == '(') {
        match('('); E();
        match(')');
    }
    else error("syntax error");
}

Input:
(1+y)*34-z

Output:
pushcon 1
push y
add
pushcon 34
multiply
push z
subtract