CIT 3136 – Lecture 7

Top-Down Parsing

Chapter 4: Top-down Parsing

A top-down parsing algorithm parses an input string of tokens by tracing out the steps in a leftmost derivation. Such an algorithm is called top-down because the implied traversal of the parse tree is a preorder traversal.

Two basic kinds of top-down parser:

- ***Predictive parsers*** **that try to make decisions about the structure of the tree below a node based on a few lookahead tokens (usually one!). This is a weakness, since little program structure has been seen before predictive decisions must be made.**
- ***Backtracking parsers*** **that solve the lookahead problem by backtracking if one decision turns out to be wrong and making a different choice. But such parsers are slow (exponential time in general).**

Fortunately, many practical techniques have been developed to overcome the predictive lookahead problem, and the version of predictive parsing called *recursive-descent* is still the method of choice for hand-coding, due to its simplicity.

But because of the inherent weakness of top-down parsing, it is not a good choice for machine-generated parsers. Instead, more powerful *bottom-up* parsing methods should be used (Chapter 5).

Recursive-descent parsing

Simple, elegant idea: use the grammar rules as recipes for procedure code. Each non-terminal corresponds to a procedure. Each appearance of a terminal in the rhs of a rule causes a token to be matched. Each appearance of a non-terminal corresponds to a call of the associated procedure.

Example

Grammar rule:

factor $\rightarrow$ **(*exp*) | *number***

Code:

```
void factor(void)
{ if (token == number) match(number);
  else {
    match('(');
    exp();
    match(')');
  }
}
```

Example, continued (2)

Note how lookahead is not a problem in this example: if the token is *number*, go one way, if the token is '(' go the other, and if the token is neither, declare error:

```
void match(Token expect)
{ if (token == expect)
    getToken();
   else error(token,expect);
}
```

Example, continued (3)

A recursive-descent procedure can also compute values or syntax trees:

```
int factor(void)
{ if (token == number)
  { int temp = atoi(tokStr);
    match(number); return temp;
  }
  else {
    match('('); int temp = exp();
    match(')'); return temp;
  }
}
```

Errors in Recursive-descent are tricky to handle:

If an error occurs, we must somehow gracefully exit possibly many recursive calls. Best solution: use exception handling to manage stack unwinding (which C doesn't have!).

But there are worse problems: left recursion doesn't work!

Left recursion is impossible!

exp* → *exp addop term* | *term

```
void exp(void)
{ if (token == ??)
  { exp(); // uh, oh!!
    addop();
    term();
  }
  else term();
}
```

EBNF to the rescue!

exp $\rightarrow$ ***term*** **{** ***addop term*** **}**

```
void exp(void)
{ term();
  while (token is an addop)
  { addop();
    term();
  }
}
```

This code can even left associate:

```
int exp(void)
{ int temp = term();
  while (token == '+'
         || token == '-')
    if (token == '+')
    { match('+'); temp += term();}
    else
    { match('-'); temp -= term();}
  return temp;
}
```

Note that right recursion/assoc. is not a problem:

exp → ***term*** **[** ***addop exp*** **]**

```
void exp(void)
{ term();
  if (token is an addop)
  { addop();
    exp();
  }
}
```

Solving the lookahead problem in greater generality:

- **Compute "First" sets for choices: First(α) = the set of tokens that can appear at the front of α. Then if we have a grammar rule $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ and First(α_i) $\cap$ First(α_j) is empty for all *i* and *j*, then a decision on a choice can be made.**
- **A problem occurs if an α_i can disappear; then we must compute "Follow" sets (the tokens that can "follow" a symbol), and show that First $\cap$ Follow is empty.**

Error Recovery in Parsers

- **A parser should try to determine that an error has occurred *as soon as possible*. Waiting too long before declaring error means the location of the actual error may have been lost.**
- **After an error has occurred, the parser must pick a likely place to resume the parse. A parser should always try to parse as much of the code as possible, in order to find as many real errors as possible during a single translation.**

Error Recovery in Parsers (continued)

- **A parser should try to avoid the error cascade problem, in which one error generates a lengthy sequence of spurious error messages.**

- **A parser must avoid infinite loops on errors, in which an unending cascade of error messages is generated without consuming any input.**

“Panic Mode” in recursive-descent

- Extra parameter consisting of a set of *synchronizing tokens.*
- As parsing proceeds, tokens that may function as synchronizing tokens are added to the synchronizing set as each call occurs.
- If an error is encountered, the parser scans ahead, throwing away tokens until one of the synchronizing set of tokens is seen in the input, whence parsing is resumed.

Example (in pseudocode)

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { EOF }) do
      getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
     error ;
     scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

Example (in pseudocode, cont)

```
procedure exp ( synchset ) ;
begin
  checkinput ( { ( , number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
        match (token) ;
        term ( synchset ) ;
    end while ;
  checkinput ( synchset, { ( , number }) ;
  end if;
end exp ;
```

Example (in pseudocode, concl.)

```
procedure factor ( synchset ) ;
begin
  checkinput ( { ( , number }, synchset ) ;
  if not ( token in synchset ) then
    case token of
    ( :    match(( ) ; exp ( { )} ) ; match( )) ;
    number : match(number) ;
    else error ;
    end case ;
  checkinput ( synchset, { ( , number }) ;
  end if ;
end factor ;
```

C-Minus and recursive-descent parsing problems:

18. *expression* → *var* **=** *expression* | *simple-expression*
19. *var* → ***ID*** | ***ID*** **[** *expression* **]**
20. *simple-expression* →
    additive-expression relop additive-expression
    | *additive-expression*
21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → **+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* → ***** | ***/***
26. *factor* → **(** *expression* **)** | *var* | *call* | ***NUM***
27. *call* → ***ID*** **(** *args* **)**