Syntax Analyzer

Chris Johnson

October 24, 2009

1 Introduction

After source code has been scanned, it must be undergo syntax analysis. The tokens generated by the lexical analyzer are then organized into an abstract syntax tree, which verifies that the program is potentially meaningful.

In this project, we construct a parser with syntax error checking.

2 Methodology

As in the lexical analyzer, we are employing Haskell to implement our parser. Once again the use of a pure functional language presents certain challenges, but also allows some tasks to be done in an intuitive, readable fashion.

There are essentially three parts to this project. First, the hand-executed algorithms to produce an LL(1) grammar, first and follow sets, and the parse table. Then the grammar must be translated into a series of functions. Finally, syntax error checking must be introduced.

2.1 Manual Preparation

The coding of this project requires three steps, and while they need not be completed before the coding begins, they ease the process significantly.

2.1.1 Grammar Massage

We are given a context-free grammar, and must generate an equivalent LL(1) grammar, so as to avoid infinite recursion (or, more likely, stack faults) for left-recursive constructs.

In the first appendix, we present the grammar at each stage of massage, starting with the unmassaged grammar.

Note that two of these steps were added for simplification purposes, and were not actually specified, but were seen to be effective in practice.

Remove ϵ-Productions The first stage of the grammar massage is the removal of ϵ-productions, or variables which reduce empty strings.

To do this, we find any variables which so reduce, find any productions which include these variables, and make two versions of that production: one with the variable removed, and one unchanged. The nullable production is removed from the grammar.

As a simple, demonstrative example, consider the grammar:

$$A \to \alpha B \gamma$$
$$B \to \beta | \epsilon$$

Where α, β and, γ are terminals. The removal of the ϵ-production from this grammar would result in:

$$A \to \alpha\gamma | \alpha B \gamma$$
$$B \to \beta$$

Eliminate Single-Production Variables Some productions initially had the form

$$A \to \alpha | \epsilon$$

and now have a single production. It eliminates a function call to replace A throughout the grammar with α.

Eliminate Left Recursion The removal of ϵ-productions was done to prepare for this stage. The goal here is to avoid entering into a loop as a terminal repeatedly brings us to the same production.

Thus, if we have a production of the following form:

$$A \to A\alpha_1 | ... | A\alpha_m | \beta_1 | ... | \beta_n$$

We replace it with the following, equivalent, productions:

$$A \to \beta_1 A' | ... | \beta_n A' | A'$$
$$A' \to \alpha_1 A' | ... | \alpha_m A' | \epsilon$$

Though, generally, this is not sufficient to eliminate left recursion, in the case of our grammar, there is no deeper recursion, so additional steps are not needed.

Left Factor We now have an unambiguous grammar, but we cannot yet make our choices based on a single token at a time. Our grammar must undergo a process called *left factoring*, wherein we combine productions with common prefixes, and insert a new variable where they diverge.

Consider:

$$A \to \alpha\beta | \alpha\gamma | \delta | \zeta | ...$$

We can produce two productions which look at a single token for their decisions like so:

$$A \to \alpha A' | \delta | \zeta | ...$$
$$A' \to \beta | \gamma$$

Deduplicate These processes described above, in practice, produce some identical productions (or parts of productions). This would imply duplicated code, so in a couple instances, I deduplicated the code.

Two such situations appeared. The first took the form:

$$A \to \alpha\beta|\beta$$

The resulting productions were:

$$A \to \alpha A'|A'$$

$$A' \to \beta$$

Obviously, this should be used judiciously. If β is trivial, then it merely adds an extra function call at the expense of code which is no simpler.

The other situation took the form:

$$A \to \alpha A'$$

$$A' \to \alpha A'|\epsilon$$

Again, assuming α isn't trivial, code can be simplified by replacing the latter production with

$$A' \to A|\epsilon$$

2.1.2 First and Follow Sets

For any CFG, each production has a set of tokens that it might accept. For an LL(1) grammar, this set is unique to a variable and a corresponding production. That is to say, given a variable, and a terminal, we can choose a production of that variable to follow, with no ambiguity.

The set of terminals that a production can accept is known as its *first set*.

Furthermore, through analysis of the grammar, and total knowledge of the first sets, we can derive the follow set of each variable, or the set of characters which might, after the variable has been fully expanded, appear next.

These are useful in constructing the parse table and determining synchronizing tokens.

These sets have been included as Appendix 2.

2.1.3 Parse Table

The parse table is a table which associates the productions of our massaged grammar with a $(variable, terminal)$ pair such that attempting to expand $variable$, given $terminal$, requires we follow the associated production.

To construct this, each variable's first set is examined, and the correct productions are associated. In the event of nullable productions, the follow set is associated with that production.

See Appendix 3 for the resulting parse table.

2.2 Implementation

The following shall be organized in the order of implementation. For the sake of narrative, a discussion of the data structures involved is reserved for last, though it may be useful to familiarize yourself with them, first.

2.2.1 Grammar

The creation of the grammar was a simple task. Haskell allows us to create new infix operators (among other things), so to make ordering sensible, I created a reverse applicator:

```
($>) = flip ($)
```

Rather than present a rule, I shall simply show an example of production and its corresponding function.

$$term \rightarrow factor\ term'$$

translates to

```
term e = e $> factor $> term'
```

I chose a particularly simple example for ease of viewing. To see how we handle multiple productions, have a look at the Appendix.

ϵ-Productions It may or may not be clear by the use of the `$` operator that each function in the production must have the same type as the function itself, which, incidentally is `State -> State`. That is, each production takes in a `State`, and returns a `State`. So an ϵ-production, not consuming any tokens, simply returns its argument.

This is why I chose `e` as my argument: to increase the intuitive readability of my grammar.

Matching A match is essentially a single-token production. It thus takes a `State`, verifies that the current token matches the expected symbol, and returns either a `State` updated so that we are looking at the next token, or replaces the current token with a `SYNTAXERR` token. Which brings us to error checking.

2.2.2 Error Checking

The additional demands of our parser[1] require that we discover errors and recover from them. The prescribed method is to go into "panic mode", consuming tokens until a synchronization character is encountered. In a functional language, it's difficult to interrupt the flow of the program to synchronize, so we have to make our parser behave politely until we run into a synchronization point.

Thus, a failed match will take the current token, and the expected symbol(s), and place them into a `SYNTAXERR` token. Any subsequent failed matches, which would be all of them,

[1] Namely, giving more descriptive errors than "SHIT DONE GONE WRONG."

as we never expect a syntax error, simply return the same state they are given. At the end of each production set[2], we check for failed matches, and call the `resolveErr` function. This takes a list of synch symbols, which is almost always the follow set for the variable in question, as well as the current `State`. It then inspects each token following the syntax error, until it finds one that matches the synch set. Once resolved, the syntax error is moved past, and the matching token is placed into "focus", so it should match whatever comes next.

Additionally, some tokens, semi-colons (";") in particular, make good synchronization tokens, and it may be tedious to wait for the next `resolveErr` function to execute, so there is also the special form of the `match` function, called `matchSynch`. If there is no syntax error, it behaves exactly like `match`. Otherwise, it calls `resolveErr`, with the supplied token as the synch set.

2.2.3 I/O and Error Reporting

Thus far, the entire program has been written in a purely functional, hermetic seal. To expose it to the real world, and the dangers of faulty programs, I introduced a pair of functions which read a file, split it into lines, runs it through a scanner, retrieves the token list and symbol table, creates a `State`, runs it through the grammar, and reports the results. Here they are:

```
main = do
        input <- getContents
        results <- return . parse . lines $ input
        mapM (putStrLn) (report results)

parse input = let (tab,toks) = scan input in
program (State (tapify' end toks) tab)
```

The `main` function is the part that actually attaches the real world to our parser, but the `parse` function does a couple of the important steps, so I thought I'd include it for fairness' sake.

Reporting The reporting is quite simple. It runs through the tokens for a lexical or syntax error, and returns a list of strings, one for each error found. Since syntax errors can actually contain lexical errors, the tokens skipped during panic mode are run through the reporter, as well.

2.2.4 Data Structures

The data structures involved are fairly simple, in concept, but they nest. Thus, I shall start from the simplest and work my way out to the one on which we perform our operations.

[2]With the exception of those which have only one production.

Line

```
data Line = Line Int String | NoLine
```

The line is simply a line of input, with a corresponding integer. The NoLine constructor is useful for associating with the EOF token.

Symbol

```
data Symbol = WHITESPACE | ... | LEXERR LexErrType String
```

A symbol is a terminal symbol or lexical error. This includes `assignop`s, `int`s, and the like. Symbols with one meaning take no arguments, while polymorphic symbols contain the string that was identified as having that type. Lexical errors include the lexical error type, as well as the string.

Haskell allows you to define equality any which way, so I chose to include some dummy symbols that would match all of a certain type without matching their arguments. A polymorphic type with string "_" would match any of that type, and some lexical errors matched the types their strings *nearly were*, to avoid spurious parser errors. (For example, `LEXERR LONGID _ == ID "_"`.)

Token

```
data Token = Token {line :: Line, sym :: Symbol}
           | SYNTAXERR {valid :: [Symbol], skip :: [Token]}
```

A token is simply a symbol, and the line it came from (for error reporting). It also is where we keep track of syntax errors, which contain a list of valid symbols (symbols which might have matched something at that point in the grammar) and a list of tokens skipped between the syntax error and the synchronization. (If this is confusing, see the section on error checking. You may want to finish reading about the data structures, first, though.)

State

```
data State = State {tape :: (Tape Token), table :: [Symbol] }
```

State is simply the currency of the parser. Each function takes it in, perhaps modifies it, and passes it on to its children, and then back up to its parent.

It carries a tape of tokens, which is simply a way of storing a list of tokens so that we can efficiently examine each in turn without losing those we've already consumed. (Incidentally, this was used to much greater effect in the lexical analyzer, as the NFA consumed a tape of characters, cutting off tokens as it finished matching.)

It also holds the symbol table, which is simply a list of symbols. (`ID`s and `LEXERR LONGID`s, for those following along at home.)

3 Conclusions

This project presented some interesting challenges with regard to function composition and state representation, but once solutions were settled on, everything else fell into place nicely. The parser reasonably represents the grammar from which it was derived, and the error handling is neatly modular, never intruding upon the program more than strictly necessary.

A Error-Free Test Program

This program should test every production with no errors.

A.1 Source

```
program example(input, output);
var x: integer;
var y: integer;
var z: real;
var a: array[0 .. 1] of integer;

function gcd(a: integer; b: integer): integer;
        begin
                if b = 0
                then
                        gcd := a
                else
                        gcd := gcd(b, a mod b)
        end;

function lcm(a: integer; b: integer): integer;
        var c: integer;
        function useless: integer;
                begin
                end;
        begin
                c := gcd(a, b);
                lcm := a * b / c
        end;

begin
        x := gcd(123,456);
        y := lcm(12,34);
        z := 3 * -1.123e45;

        i := 10;
```

```
        while i > 0 do
                begin
                        i := i - 1
                end;

        if not (i = 0) then
                i := 1
        else
                i := 0;

        a[0] := 1;
        a[1] := a[0]
end.
```

A.2 Output

```
$ ./Parser <test.pas
```

B Error-imbued Test Program

This test program is nearly identical to the previous one, except designed to cause every lexical error.

B.1 Source

```
program example(input, output);
var x: integer;
var y: integer;
var z: real;
var a: array[0 .. 3] of integer;
var thisidisreallytoolong: integer;

><

function gcd(a: integer; b: integer): integer;
begin
if b = 0
then
gcd := a
else
gcd := gcd(b, a mod b)
end;

function lcm(a: integer; b: integer): integer;
```

```
var c: integer;
function useless: integer;
begin
end;
begin
c := gcd(a, b);
lcm := a * b / c
end;

begin
x := gcd(123,456);
y := lcm(12,34);
z := 3 * -1.123e45;

i := 10;
while i > 0 do
begin
i := i - 1
end;

if not (i = 0) then
i := 1
else
i := 0;

a[0] := 112358132134;
a[1] := 2718281828459e-12;
a[2] := .314159265358979e1;
a[3] := 1.23e456
end.
```

B.2 Output

```
$ ./Parser <errors.pas
Lexical Error: Received Line 6: thisidisreallytoolong (Extra Long Identifier)
Syntax Error:  Received Line 8: "><"; Expected 'var', 'function', or 'begin'
Lexical Error: Received Line 8: >< (Unrecognized Symbol)
Lexical Error: Received Line 45: 112358132134 (Extra Long Integer)
Lexical Error: Received Line 46: 2718281828459e-12 (Extra Long Whole Part)
Lexical Error: Received Line 47: .314159265358979e1 (Extra Long Fractional Part)
Lexical Error: Received Line 48: 1.23e456 (Extra Long Exponent)
```