Compiler

Dr. Yihsiang Liow (March 28, 2020)

Contents

hello world

Chapter 1

Grammar

1.1 Grammar

1.2 Derivation

1.3 Parse tree

Chapter 2

Recursive descent parser

2.1 Recursive descent parser

Recall that a parser builds a derivation from a stream of tokens based on a given grammar. Builda a derivation of course builds a parse tree.

A **recursive descent parser** is a parser that builds the derivation in the following way:

- It starts with the start symbol
- A production rule is chosen in the following way:
    - Given a sentinel form, it selects the leftmost variable V for substitution.
    - The produce rule $V \to w$ is chosen from the rules so that the leftmost symbol of w is the next expected terminal to derive or it is a variable that can in turn derive the next expected terminal.

For instance consider the following grammar

$$
\begin{aligned}
&S \to \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S\\
&S \to \texttt{begin}\ S\ L\\
&S \to \texttt{print}\ \texttt{num}\ ;\\
&L \to\ ;\ S\ L\\
&L \to \texttt{end}\\
&E \to \texttt{num == num}
\end{aligned}
$$

Consider the following stream of tokens

```
if num(1) == num(2) then
begin
    if num(3) == num(4) then
        print num(1000)
end
else
begin
    print num(2000);
    print num(3000)
end
```

Here's the derivation of the token stream:

$$\begin{aligned}
\underline{S} &\implies \texttt{if } \underline{E} \texttt{ then } S \texttt{ else } S \\
&\implies \texttt{if num == num then } \underline{S} \texttt{ else } S \\
&\implies \texttt{if num == num then begin } \underline{S}\ L \texttt{ else } S \\
&\implies \texttt{if num == num then begin if } \underline{E} \texttt{ then } S \texttt{ else } S\ L \texttt{ else } S
\end{aligned}$$

Etc.

Let me do it slowly, so that you see the process.

STEP 1. Since S is the start symbol, I have to start with

$$\underline{S} \implies$$

The token stream is

```
if ...
```

The *leftmost* token is `if`. At some point in the derivation, you *have* to spit out an `if` as the first token. Then I notice the production rule

$$S \to \texttt{if } E \texttt{ then } S \texttt{ else } S$$

is able to produce an `if` token on the *left*. In fact it's the only rule that looks like

$$S \to \texttt{if } \ldots$$

STEP 2. So now I have

$$\underline{S} \implies \texttt{if } E \texttt{ then } S \texttt{ else } S$$

This means (removing the `if`), I need to continue with

$$E \texttt{ then } S \texttt{ else } S \implies$$

to derive (note that I removed the beginning `if`)):

```
   num(1) == num(2) then
begin
    if num(3) == num(4) then
        print num(1000)
end
else
begin
    print num(2000);
    print num(3000)
end
```

Then I noticed that the E (the leftmost variable of the following):

$$E \texttt{ then } S \texttt{ else } S \implies$$

can derive the `num(1)` token:

$$E \to \texttt{num == num}$$

(in fact it derives the next two tokens as well). There is no other production for E. Therefore I have the following derivation

$$E \texttt{ then } S \texttt{ else } S \implies \texttt{num(1) == num(2) then } S \texttt{ else } S$$

Removing what is already derived, basically I have to continue with

$$\texttt{then } S \texttt{ else } S \implies$$

to derive

```
                     then
begin
    if num(3) == num(4) then
        print num(1000)
end
else
begin
    print num(2000);
    print num(3000)
end
```

STEP 3. From this step in the derivation

$$\texttt{then } S \texttt{ else } S \implies$$

I have to derive the `then` in

```
                    then
begin
    if num(3) == num(4) then
        print num(1000)
end
else
begin
    print num(2000);
    print num(3000)
end
```

Obviously there's nothing to do – I'm going to remove `then`.

STEP 4. Now I need to continue the derivation of

$$S \texttt{ else } S \implies$$

to derive

```
begin
    if num(3) == num(4) then
        print num(1000)
end
else
begin
    print num(2000);
    print num(3000)
end
```

Etc. Get it?

That is the main idea behind recursive descent parsing. To summarize, if I'm continuing the derivation with

$$\underline{V} w_0 \implies$$

where V is a variable to derive the remaining token stream

$$t w_1$$

where t is a token, then I need to quickly find a production rule of the form

$$V \to t w_2$$

But ... be careful: What if there's a rule

$$V \to \epsilon$$

Then t might be derived by *another* variable after V, i.e., a variable in w_0. Another thing to note is that in the above example, I can read off very quickly what is the leftmost terminal that a variable can derive by looking at a production rule. For instance for the rule

$$\underline{S} \implies \texttt{if } \underline{E} \texttt{ then } S \texttt{ else } S$$

I see easily that S can derive `if` (on the left). But it's possible that the leftmost terminal derived by a variable occurs after multiple derivations. For instance consider

$$A \to BC$$
$$B \to bAc$$

In this case A can derive a leftmost terminal of b, not in one derivation but in two:

$$\underline{A} \implies \underline{B}C \implies bAcC$$

Note that it's possible to have *two* possible candidate product rules. In this case, we have a **conflict**. The recursive descent parser that looks at the next token to derive will not be able to parse the token stream. The recursive descent parser that looks at just one token to be derived is called an $LL(1)$ parser. In general an LL parser is a parser that operates this way:

- L – analyze tokens left-to-right
- L – choose the leftmost variable for substitution

Sometimes conflicts in a $LL(1)$ parser can be resolved when you look ahead by *two* tokens. If you look at two tokens to decide which production rule to choose (when there's a conflict), then the recursive descent parser is called an $LL(2)$ parser. In general we have $LL(k)$ which looks at k (or less) tokens in the token stream to determine which production rule to use.

If there is always a unique production rule to use in an $LL(k)$ parser, then the runtime is linear (assuming it takes constant time to look for the right production rule). If the grammar allows you to choose a unique production rule after looking at at most k tokens, we say the grammar is an $LL(k)$ grammar.

If there are say two possible production rules when you look at k tokens, you can try both. In general you can talk about $LL(k)$ recursive descent parser with backtrack when there are multiple production rules after looking at k tokens. In this case the runtime is exponential. The k is sometimes called the **look aheads** because you are looking into the token stream to see what's coming.

Let's stick to non-backtracking recursive descent for now.

So to speed up the parsing process, I need to have a quick way to answer the following question: Given a variable V and a token t, find all production rules that look like this:

$$V \to t...$$

This is the first step (a pre-processing step) in recursive descent parsing. And we hope that this can be done quickly. If there is a unique production rule for all occurring V, t, then the grammar is $LL(1)$ and we can write a recursive descent parser that looks at only one token (look ahead is 1) in the token stream.

In particular if you look at the above example grammar: For instance consider the following grammar

$$\begin{aligned}
S &\to \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
S &\to \texttt{begin}\ SL \\
S &\to \texttt{print}\ \texttt{num}\ \texttt{;} \\
L &\to \texttt{;}\ S\ L \\
L &\to \texttt{end} \\
E &\to \texttt{num}\ \texttt{==}\ \texttt{num}
\end{aligned}$$

All the possibly pairs of V, t looks like this:

$$\begin{aligned}
(S, \texttt{if}) &: \text{choose } S \to \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
(S, \texttt{begin}) &: \text{choose } S \to \texttt{begin}\ SL \\
(S, \texttt{print}) &: \text{choose } S \to \texttt{print}\ \texttt{num}\ \texttt{;} \\
(S, \texttt{;}) &: \text{choose } L \to \texttt{;}\ S\ L \\
(S, \texttt{end}) &: \text{choose } L \to \texttt{end} \\
(E, \texttt{num}) &: \text{choose } E \to \texttt{num}\ \texttt{==}\ \texttt{num}
\end{aligned}$$

This is a home run: no conflicts.

Now let's write an OCAML program to parse words in this grammar.

The input to the parser is a stream of tokens. For simplicity, I'll use a list of tokens. Here's my type for tokens:

```
type token = Num of int
           | If
           | Then
           | Else
           | Begin
           | End
           | Print
           | IsEqu
           | Semicolon
;;
```