Unit 5: Recursive Descent and LL(1) Grammars - Part One

SCC 312 Compilation

Barry Porter

A Reminder

- LL(1) :
  - Left-to-right token processing
  - Leftmost derivation
  - Top-down parsing
  - One lookahead token

Aims

- What is a recursive descent parser?
    - How does it work? Why does it work?
- How to process a non-terminal
- Illustrated by a small worked example
- What kind of error reporting and recovery is possible?

Recursive Descent and LL(1) Grammars

- We start with a top-down strategy for writing parsers called ***recursive descent***
  - Also known as ***predictive recursive descent***

- It is useful particularly for hand-generation of a compiler from a grammar

Recursive Descent Parsers

- The parser is going to consist of a collection of methods
  - One for each non-terminal of the grammar
  - Named after that non-terminal
  - With its method body derived semi-automatically from the grammar rule for that non-terminal

Motivation

- Consider the **<if statement>** method
- It is called when the next token is the word “if”
- We are at the start of a sequence of tokens which represents an **if** statement
- We want our method to find its way to the end of this sequence, ready to look at the first token of whatever follows the **if** statement
- **<if statement> ::= if <expression> then <statement> fi**

Motivation

- So we want it to find its way over
  - The token “if”
  - The tokens representing <expression>
  - The token “then”
  - The tokens representing <statement>
  - Etc.
- **<if statement> ::= if <expression> then <statement> fi**

Motivation

- This sequence is precisely specified by the **<if statement>** grammar rule
- **<if statement> ::= if <expression> then <statement> fi**
- To find our way over the tokens of **<expression>,** we must call the method corresponding to **<expression>,** etc.

General Overview of the Parser

- The method for a particular non-terminal X is called when the parser has decided that it wants to recognise an X starting at this point in the input stream
- The method will “consume” the tokens making up the X, and leave the parser ready to process the first terminal of the next non-terminal in the input stream
  - As a by-product, the method will also build the appropriate piece of the parse tree, and whatever else is required for X

General Overview of the Parser

- The parser has to know at each point what non-terminal it wants to recognise
- It does this by being allowed to look at the next token in the input stream
  - That is, the first token of the non-terminal it is deciding to recognise

General Overview of the Parser

- The parser needs to know all the possible first terminals or tokens for each non-terminal
- To make this work, we see that there are restrictions on how these sets of terminals overlap
- And these restrictions are the requirements for a grammar to be LL(1)

Structure of the Parser

- We have a set of methods to recognise the various elements of the grammar, one for each non-terminal
- There is a variable **nextSymbol**, which contains the next token recognised by the lexical analysis phrase

Structure of the Parser

- The method corresponding to non-terminal X expects to find in **nextSymbol** one of the tokens listed in FIRST(X)
- It expects to finish by leaving in **nextSymbol** one of the tokens which is in FIRST(Y) for some Y which can appear immediately after X
    - That is, it is a token in the set FOLLOW(X), also known as the *Lookahead* or Follow Set

The acceptTerminal Method

- We have a method

```
 acceptTerminal (t):
```

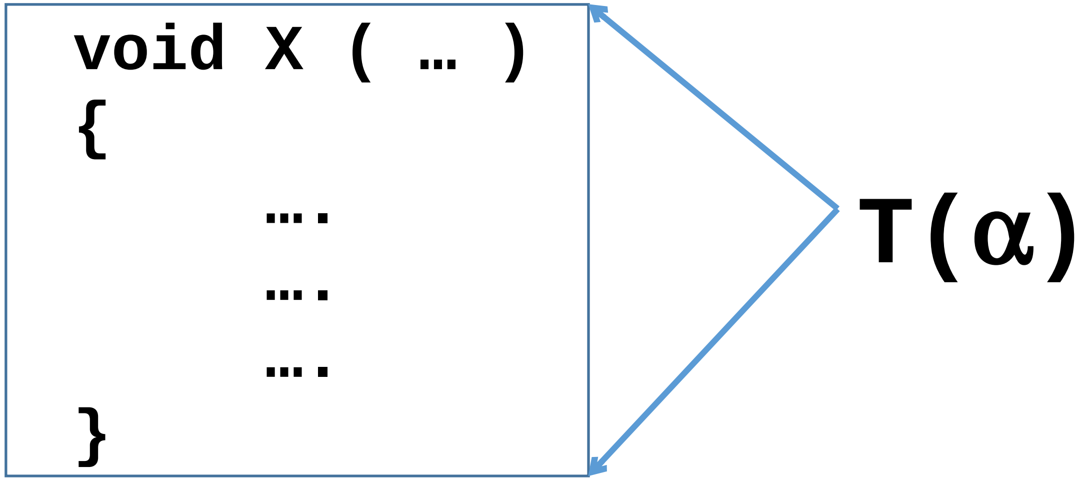
't' is the token we now expect

```
if ( nextSymbol == t )
    get next token from lexical
             analyser into nextSymbol ;
else
     report error ;
```

Non-terminal <X>

- We have one method for each non-terminal
- Suppose we have non-terminal X, and its grammar rule is <X> ::= α …
- … then we have a method named X with body T(α).
  - What could α be? In other words, what can appear on the RHS of a production rule?
  - There are four possibilities, shown next.

- <X> ::= α
- a method X with body T(α).



```
void X ( … )
{
        ….
        ….
        ….
}
```

Possibilities for α

Possibility	General	Example
1. A single terminal	<X> ::= t	<X> ::= return
2. A single non-terminal	<X> ::= <NT>	<X> ::= <declarationList>
3. A sequence of terminals and non-terminals	<X> ::= a1 a2 .. aN	<X> ::= if <expression> then <statement> …
4. A set of alternatives	<X> ::= a1 | a2 | .. | aN	<X> ::= <ifStatement> | <whileStatement> | …

<X> ::= α First Possibility

- A single terminal
- **`<X> ::= t`**
- **`<X> ::= return`**
- If α is a single terminal t, then T(α) is
  - **`acceptTerminal (t);`**
  - if α is "return", T(α) would be "**`acceptTerminal (returnSymbol);`**"

<X> ::= α Second possibility

- A single non-terminal
- `<X> ::= <NT>`
- `<X> ::= <declarationList>`
- If α is a single non-terminal NT, then T(α) is
  - `NT() ;`
- That is, a call of the method associated with non-terminal Y
  - if α is " `<declarationList>` "
    T(α) would be "`declarationList () ;` "

<X> ::= α Third Possibility

- A sequence of terminals and non-terminals
- `<X> ::= a1 a2 .. aN`
- If α is a sequence of terminals and non-terminals a1 a2 … an, then T(α) is the sequence:

```
T(a1) ;
T(a2) ;
...
T(an) ;
```

<X> ::= α Third Possibility: example

- `<X> ::= if <expression> then <statement> …`
- If a is "**`if <expression> then <statement> …`** ",
  T(α) would be

```
acceptTerminal (ifSymbol) ;
expression() ;
acceptTerminal (thenSymbol) ;
statement() ;
…
```

<X> ::= α Fourth Possibility

- A set of alternatives
- `<X> ::= a1 | a2 | .. | aN`
- If α is a set of alternatives a1 | a2 | … | an, then T(α) is

```
switch (nextSymbol)
{
    case FIRST(a1) :
        T(a1) ;  break ;
    case FIRST(a2) :
        T(a2) ;  break ;
        ...
    case FIRST(an) :
        T(an) ;  break ;
} // end of switch
```

<X> ::= α Fourth Possibility

- `<X> ::= <ifStatement> | <whileStatement> | …`
- If a is "<ifStatement> | <whileStatement> | …", then T(a) is

```
switch (nextSymbol)
{
    case ifSymbol :
      ifStatement() ;  break ;
    case whileSymbol :
      whileStatement() ; break ;
    ...
} // end of switch
```

Dealing with Extended BNF

- Some versions of BNF are extended, so that {x} means zero or more repetitions of x
- Then this would be transformed to:

  -
    ```
    while (nextSymbol is in FIRST(x))
    {
                T(x) ;
    } // end of while
    ```
- For an example, see “expression” and “term” later.

Dealing with Extended BNF

- Similarly [x] means zero or one occurrence of x in some extensions of BNF
- Then this would be transformed to:

  -
    ```
    if  (nextSymbol is in FIRST(x))
        {
            T(x) ;
        } // end of if
    ```

The main Method

- The main method, to get everything started, has the body:
  -
    ```
    get first token from lexical analyser into nextSymbol ;
    <program>() ; (or whatever the distinguished symbol is)
    acceptTerminal (eofSymbol) ;
    report success ;
    ```
- We need to ensure there are no extraneous characters after the valid <program> string

The Recursive Descent Parser

An Example

- Now consider the grammar (with the obvious things as lexical tokens):

```
<statement> ::= if <expression> then
                <statement> fi |
                <variable> := <expression>
<variable> ::= ident [ ( <expression> ) ]
<expression> ::= <term> { + <term> }
<term> ::= <factor> { * <factor> }
<factor> ::= <variable> | ( <expression> )
```

The Recursive Descent Parser

```
void acceptTerminal (Token t)
{
    if (nextSymbol == t)
        get next token from lexical
           analyser into nextSymbol ;
    else
        report error ;
} //  end of method
```

The Recursive Descent Parser

<statement> ::= if <expression> then
<statement> fi |
<variable> := <expression>

```
void <statement>()
  {
  switch (nextSymbol)
      {
      case ifSymbol :
          acceptTerminal (ifSymbol) ;
          <expression>() ;
          acceptTerminal (thenSymbol) ;
          <statement>() ;
          acceptTerminal (fiSymbol) ;
          break ;
```

The Recursive Descent Parser

<statement> ::= if <expression> then
<statement> fi |
<variable> := <expression>

```
   case ident :
       <variable>() ;
        acceptTerminal (becomesSymbol) ;
       <expression>() ;
       break ;
  } // end of switch
 } // end of method
```

The Recursive Descent Parser

<variable> ::= ident [(<expression>)]

```
void <variable>()
{
    acceptTerminal (ident) ;
    if (nextSymbol == leftParenthesis)
    {
        acceptTerminal (leftParenthesis) ;
        <expression>() ;
        acceptTerminal (rightParenthesis) ;
    } // end of if
} // end of method <variable>
```

The Recursive Descent Parser

<expression> ::= <term> { + <term> }

```
void <expression>()
{
    <term>() ;

    while (nextSymbol == plusSymbol)
    {
        acceptTerminal (plusSymbol) ;
        <term>() ;
    } // end of while

} // end of method <expression>
```

The Recursive Descent Parser

<term> ::= <factor> { * <factor> }

```
void <term>()
{
    <factor>() ;
    while (nextSymbol == timesSymbol)
    {
        acceptTerminal (timesSymbol) ;
        <factor>() ;
    } // end of while
} // end of method <term>
```

The Recursive Descent Parser

<factor> ::= <variable> | (<expression>)

```
void <factor>()
{
    switch (nextSymbol)
    {
      case ident :
         <variable>() ;
         break ;
      case leftParenthesis :
         acceptTerminal (leftParenthesis) ;
         <expression>() ;
         acceptTerminal (rightParenthesis) ;
         break ;
    } // end of switch
} // end of method <factor>
```

The Recursive Descent Parser

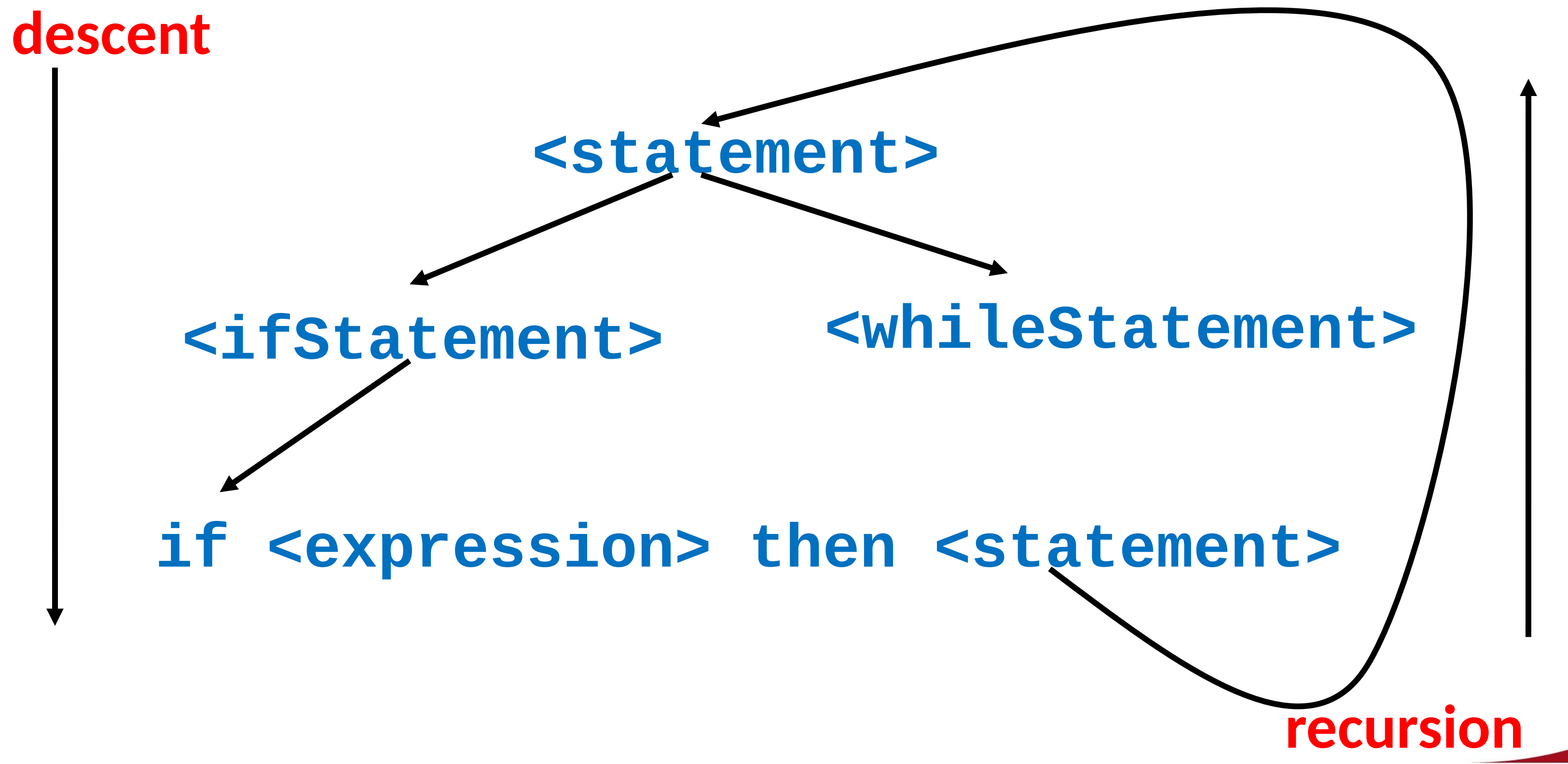
```
void parse()
{
        get first token from lexical  analyser into
nextSymbol ;

        <statement>() ;

        acceptTerminal (eofSymbol) ;

        report success ;
} // end of method parse
```

Why is it called "recursive descent"?



Error Reporting and Recovery

Error Reporting

- This parser is pretty unhelpful if the source text is syntactically invalid
- It will simply stop at the first place where there is an error, and report that there is an error
- We should at least be as helpful as possible in reporting what the error was
- In this approach, we have fairly obvious places where we can detect errors and produce appropriate messages
    - i.e. the default clause of a switch statement
    - The unused “else” clause of an if statement

Error Reporting

- So the **acceptTerminal** method should report what token it was searching for and what it found
- Each case statement should include a default alternative, which could report the token it found and what syntactic category it was considering
- In both cases the parser should print out an indication of what source text line, and where on the line, the error occurred

Error Reporting

```
void acceptTerminal (Token t)
{
  if (nextSymbol == t)
      get next token from lexical
analyser into nextSymbol ;
  else
      report error – expected t, found nextSymbol,
             at line/char ;
} //  end of method acceptTerminal
```

Error Reporting

<statement> ::= if <expression> then <statement> fi |
<variable> := <expression>

```
void <statement>()
{
    switch (nextSymbol)
    {
        case ifSymbol :
            acceptTerminal (ifSymbol); … break ;
        case ident :
            <variable>(); … break ;
        default :
            report error – expected if or ident in
              <statement>, found nextSymbol, at
            line/char;
    } // end of switch
} // end of method <statement>
```

Error Reporting

<expression> ::= <term> { + <term> }

```
void <expression>()
{
  <term>() ;
  while (true)
  {
     switch (nextSymbol)
     {
       case plusSymbol :
            acceptTerminal (plusSymbol) ; …
            break ;
       case FOLLOW(<expression>) :
           break out of loop ;
       default : report error ;
      } // end of switch
   } // end of while
} // end of method <expression>
```

Error Reporting

- One advantage of a parsing technique which avoids back-tracking is that an error is likely to be recognised somewhere close to where it actually occurred

Error Recovery

- A more sophisticated parser would try to “fix up” the source code sufficiently to be able to continue scanning the text for further errors
  - but without generating lots of spurious error messages
- A common mechanism is panic mode
  - While parsing we maintain a set of synchronising tokens
  - If the parser finds an error it scans forward until it finds one of these synchronising tokens
    - Throwing away tokens without error checking
  - Then it resumes parsing

Learning Outcomes

- You should now understand the recursive descent approach