Unit 6: Recursive Descent and LL(1) Grammars (Part Two)

SCC 312 Compilation

John Vidler
j.vidler@lancaster.ac.uk

Original slide deck by John Mariani

Aims

- How can we make sure the parser will work? what are the restrictions on LL(1) languages? what can we do if the language falls outside of those restrictions?
- What is left-recursion and why is it a problem?
- What is the definition of an LL(1) language?
- Can we handle non-LL(1) situations?

- Appendix
  - What’s the difference between parsing and recognising?

Making it work

Making it Work

- To make this parser work, it must always be able to decide what to do next
- So, if there is a grammar rule with a right-hand side a1 | a2 | ... | an, this is going to translate into a switch statement in which all the alternatives must be distinct
- In other words, the FIRST sets of all the alternatives a1, a2, ..., an in the rule must be disjoint (that is, no overlap)

The Example

- In the example the right-hand side of the first rule has FIRST sets {`ifSymbol`} and {`ident`}
- These are disjoint so the switch in the `<statement>` method is deterministic

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

A Different Example

- Suppose this was the rule.
- Then the FIRST sets for the three alternatives are {`ifSymbol`}, { `ifSymbol`} and {`ident`}
- So the parser could not make a decision in this case
- A possible way round this would be to rewrite the rule by **left-factoring**.

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

Left-Factoring

if <expression> then <statement>	fi
if <expression> then <statement>	else <statement> fi

fi

if <expression> then <statement>

else <statement> fi

<if_remainder> ::= else <statement> fi | fi

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

A Solution to the Problem

- The FIRST sets for the first rules are {`ifSymbol`} and { `ident` }, and for the second rule { `elseSymbol` } and { `fiSymbol` }
- So the problem is solved

<statement> ::= if <expression> then
<statement> <if_remainder> |
<variable> := <expression> ;

<if_remainder> ::= else <statement> fi | fi ;

A Solution to the Problem

- We have found an equivalent way of writing the grammar, but this may have ramifications later in the compiler
- For example, the different structure assigned by this rewrite may make it less easy to generate the appropriate machine code
- We could also rewrite in extended BNF :
- If <expr> then <statement> [else <statement>] fi

Left-Recursion

What it is, the problems it causes and possible solutions

Left-Recursion

- Consider a left recursive grammar rule
  - A $\rightarrow$ Av | u
- This would translate to

```
switch (nextSymbol)
{
    case FIRST(Av) :
        T(Av) ;  break ;
    case FIRST(u) :
        T(u) ;  break ;
} // end of switch
```

Left-Recursion

- $A \rightarrow Av \mid u$
- But FIRST(Av) and FIRST(u) are both { u }
- So we cannot handle left recursion directly in a recursive descent parser
- Alternatively we could write this
  - $A \rightarrow uB$
  - $B \rightarrow vB \mid \varepsilon$
- Both grammars will generate the same form of sentence : u{v}+ and so are equivalent.
- We have not yet seen what to do with a null-production

Null-Productions

- A fifth possibility
- If the right-hand side a of the grammar rule for non-terminal X has the form a1 | a2 | … | an | ε, then T(a) is

```
switch (nextSymbol)
{
    case FIRST(a1) :
        T(a1) ;  break ;
     ...
    case FIRST(an) :
        T(an) ;  break ;
    case FOLLOW(X) :
        break ;  // i.e. do nothing
} // end of switch
```

Null-Productions

A → uB

```
void A()
    { acceptTerminal (uSymbol) ; B() ; }
```

B → vB | ε

```
void B()
     { switch (nextSymbol)
           {
           case vSymbol :
              acceptTerminal (vSymbol) ;
                 B() ;  break ;
           case FOLLOW(B) :
                 break ;  // i.e. do nothing
           } // end of switch
     } // end of method B
```

Null-Productions

- We now have a new restriction on grammars for which we can write recursive descent parsers
- Where we have null-productions, as in the above example, FOLLOW(X) must be disjoint (no overlap) from all the FIRST(ai) sets
- <X> ::= a1 | a2 | … | an | ε

Our Example Grammar

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

Our Example Grammar

- We have already applied extended BNF to our grammar.
- <expression> ::= <term> { + <term> }

- Without it, we could have stated the following
- <expression> ::= <expression> + <term> | <term>

- But this is left recursive.

An Alternative Solution to Left Recursion

- <expression> ::= <term> { + <term> }
- The body of <expression> is therefore

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

An Alternative Solution to Left Recursion

- In order for this to work the parser must be certain when to iterate and when not
- The parser should iterate if the next token is in FIRST(+ <term>), which is { + }
- It should not iterate when the next token is in FOLLOW(<expression>), which is { **then**, **)**, **fi**, **EOF** }
- These are disjoint, so everything is fine
- Similarly for the other rules.
- (If you don't understand how we calculated FOLLOW(<expression>), see the appendix.

Defining an LL(1) grammar

LL(1) Grammars

- If a programming language has a grammar which is LL(1), we can write a recursive descent parser for it
- For a grammar to be LL(1), it must obey the following rules
  - We are assuming a simple BNF without extensions, but the rules could be re-formulated to deal with extended BNF

LL(1) Grammars : Conditions

- If a non-terminal has the grammar rule
  $X \rightarrow a1 \mid a2 \mid \ldots \mid an$
  then we must have
  $\text{FIRST}(ai) \cap \text{FIRST}(aj) = \varnothing$ for all $i \neq j$

- If any non-terminal X can generate the null string then we must have
  $\text{FIRST}(X) \cap \text{FOLLOW}(X) = \varnothing$

The "dangling else" problem

The "dangling else" problem

- Suppose the first rule of our example grammar was as shown below.

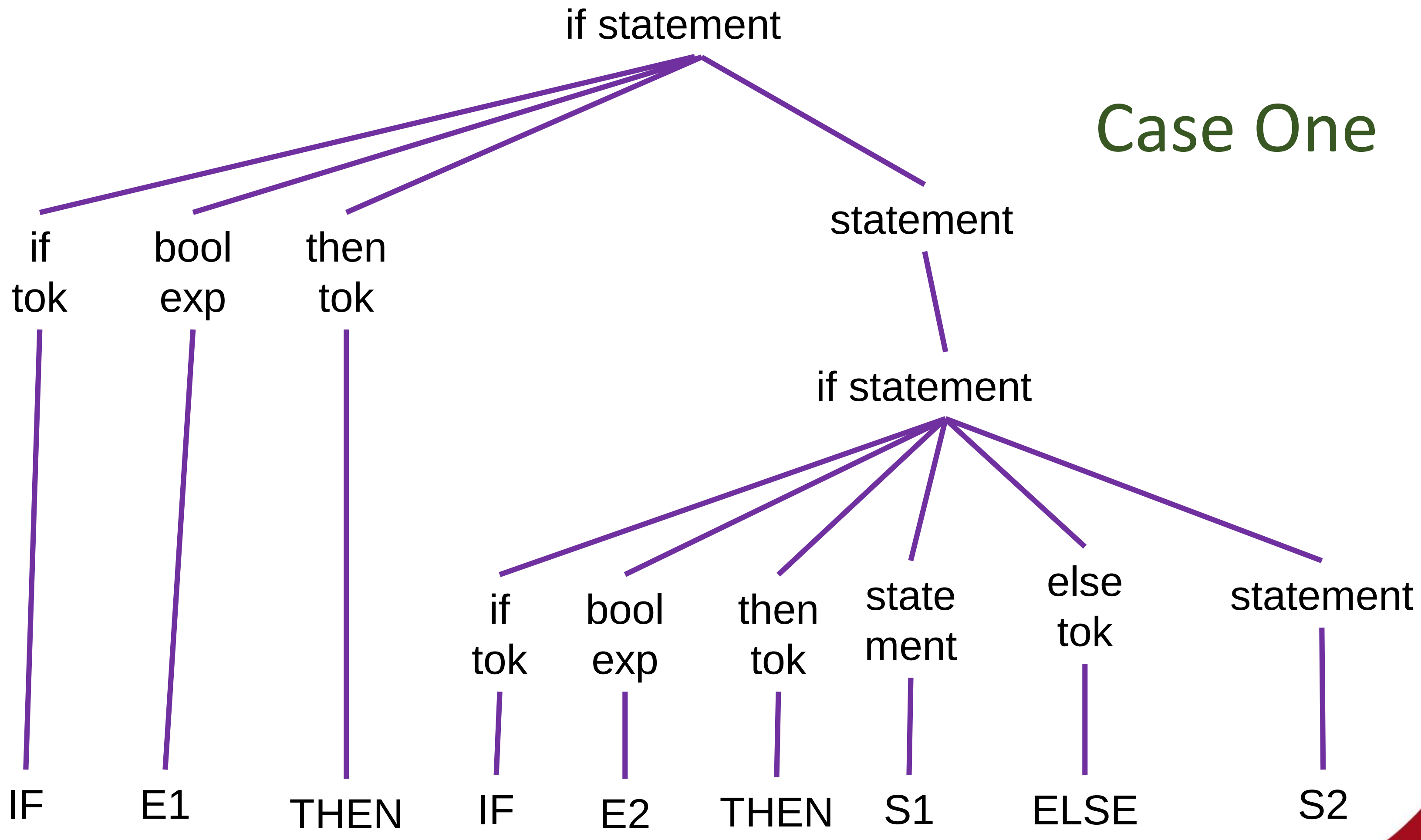
<statement> ::= <variable> := <expression> |
if <expression> then <statement>
[else <statement>]

The "dangling else" problem

- The grammar is not unambiguous - there are two parses of
  - if E1 then if E2 then S1 else S2
  - could mean
    - if E1 then { if E2 then S1 else S2 }
  - or
    - if E1 then { if E2 then S1 } else S2

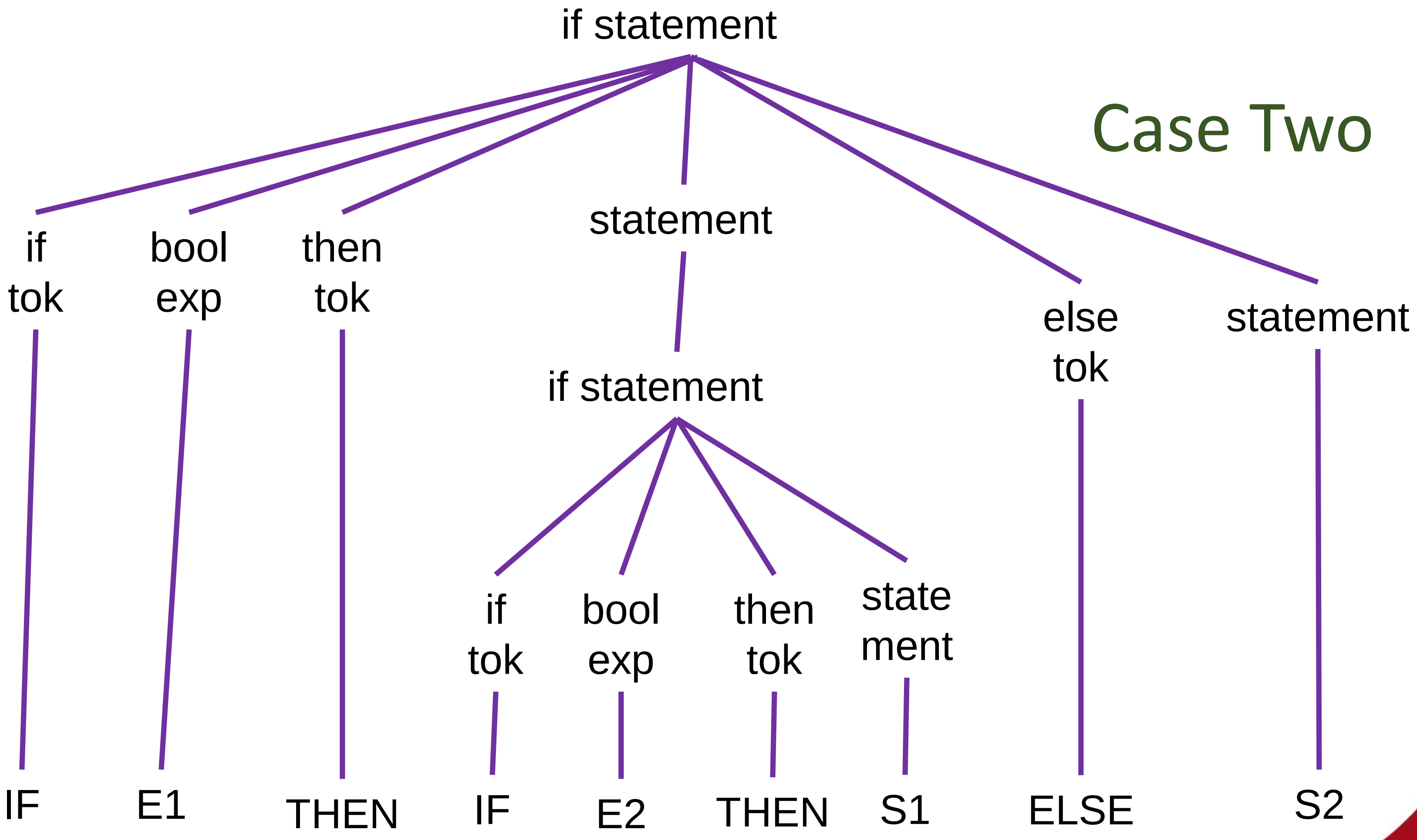
if E1 then { if E2 then S1 else S2 }

Case One



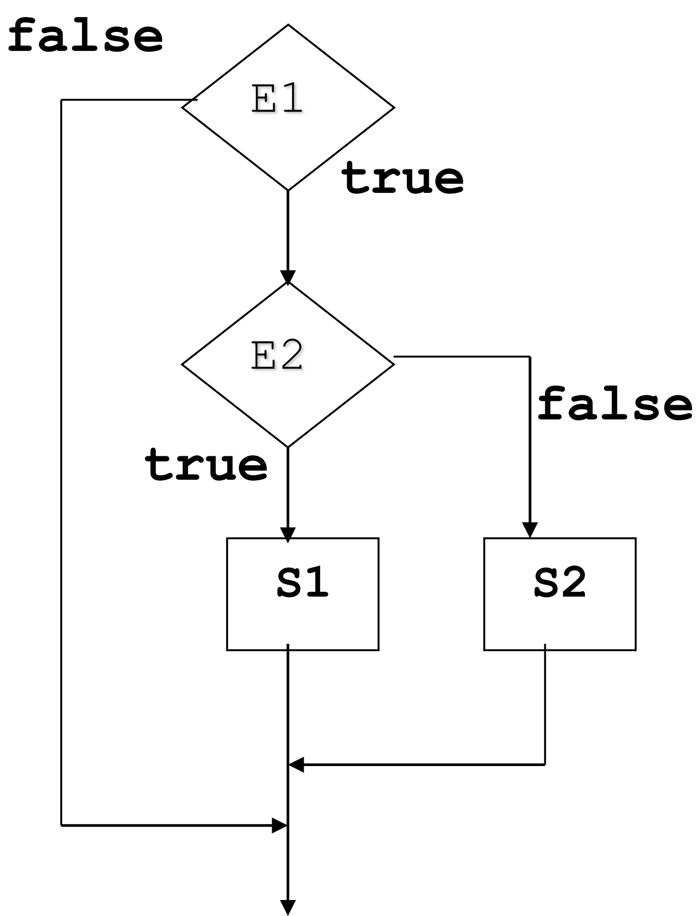
if E1 then { if E2 then S1 else S2 }

Case Two

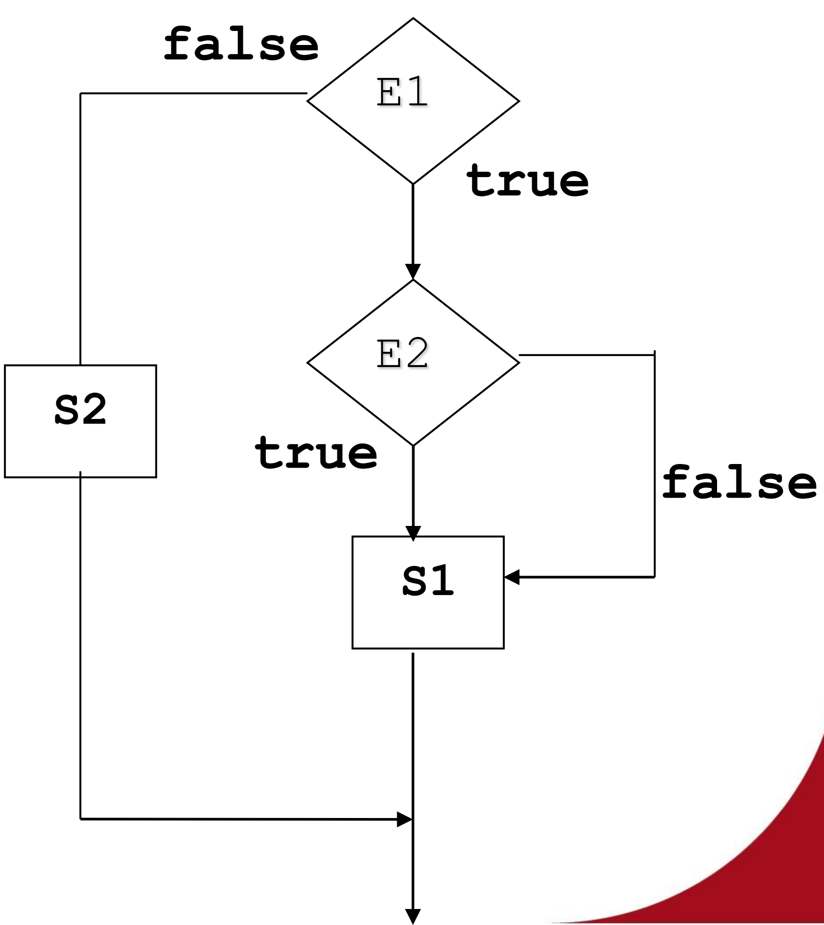


the dangling else problem

- Case One :
  if E1 then { if E2 then S1 else S2 }



- Case Two :
  if E1 then { if E2 then S1 } else S2



<statement> ::= if <expression> then <statement> [else <statement>]
| <variable> := <expression>

```
void <statement>()
{
  switch (nextSymbol)
  {
    case ifSymbol :
              acceptTerminal (ifSymbol) ;
              <expression>() ;
              acceptTerminal (thenSymbol) ;
              <statement>() ;
              if (nextSymbol == elseSymbol)
              {
                   acceptTerminal (elseSymbol) ;
                   <statement>() ;
              } // end of if
      break ;
```

<statement> ::= if <expression> then <statement> [else <statement>]
| <variable> := <expression>

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

The "dangling else" problem

- The effect of this (as in the Pascal compiler) is to assume any "dangling else" belongs to the innermost "if" (case one on the flowchart slide)
- The danger with this is that part of the syntax of the programming language is encoded in the implementation of the parser, rather than in the grammar

- Viewer's Exercise : what if we introduce a 'fi' or 'endif' to the language? does this fix the dangling else problem?

Learning Outcomes

- You should now understand
  - The definition of LL(1) grammars and their restrictions, and the practical reasons those restrictions exist
  - How to deal with left-recursion and null productions

THE END

The End

Appendix : Parsing vs. Recognising

Parsing v Recognising

- Notice also that this parser is in fact just a *recogniser*; it simply reports success or failure of an attempt to parse an input string
- It would need to be extended to do something useful (for example, built a bit of the parse tree) when it has recognised some particular non-terminal as a sequence of terminals and other non-terminals

Parsing v Recognising

- A way to do this would be for each method to pass back the piece of parse tree which it has generated

- We could rewrite the method for <variable>, for instance, to return a pointer to a Node object structure which contains
    - An indication of which rule was used to generate this node
    - Pointers to the relevant subordinate Nodes on the parse tree

Parsing v Recognising

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

```
Node <variable>()
{
  Node t1 = new Node() ;
  t1.rule = number of "simple <variable>" rule;
  t1.field1 = acceptTerminal (ident);
  if (nextSymbol == leftParenthesis)
  {
     acceptTerminal (leftParenthesis) ;
     t1.rule = number of "indexed <variable>" rule;
     t1.field2 = <expression>() ;
     acceptTerminal (rightParenthesis) ;
   } // end of if
   return t1 ;
 } // end of method <variable>
```

FOLLOW (<EXPRESSION>)

- The obvious followers are then and).

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

FOLLOW (<EXPRESSION>)

- The root is <statement> which is followed by eof. A statement can be an assignment statement, which ends in an <expression>. Therefore eof is a possible follower of <expression>.
- Similarly, the <statement> appearing before fi could be an assignment statement so once again <expression> could be followed by fi.

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