Languages and Compilers
LL(1) Specifications

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University

Today's plan

- What is LL(1) and why should you care?
- Director sets
- Microsyntax

Why LL(1)?

- We've been doing derivation sequences by hand – we're going to want our **parser** (syntax analyser) to figure out the derivation sequence automatically.

- Some language specifications are **LL(1)**.
  - i.e. we can take a BNF spec, and say whether it's LL(1) or not – and if it isn't, possibly rewrite it so it is.

- An *LL(1) specification* has certain properties that make it particularly easy to write a parser for...

What Is An LL(1) Specification?

- With an LL(1) specification every step of the derivation sequence is uniquely defined by the next terminal symbol in the input string.

L input string is scanned from left to right.

L leftmost derivation is always used (i.e. you substitute the leftmost non-terminal)

(1) only 1 symbol look-ahead is required.

Other rules like this exist – e.g. LALR(1) – **rightmost** derivation, 1 symbol **l**ook**a**head.

LL(1) In The BNF Context.

- When there is a rule of the form
  <A> ::= α | β | ... ;
- and we are looking to substitute for <A> in the next derivation step,
- then the production to apply is uniquely defined by the next terminal symbol on the input string
  - i.e. at every step in the derivation sequence, we only need to look at the **next word** to tell which rule to use

An LL(1) Specification For A Robot Movement Language.

```
<Session>      ::= <Instruction> <More> end ;
<More>         ::= <Instruction> <More> | <> ;
<Instruction>  ::= home |
                   <Direction> <Distance> |
                   <Coord> ;
<Direction>    ::= forward | back | left | right;
<Distance>     ::= Integer | <> ;
<Coord>        ::= "(" <Signed-Int> “,”
                       <Signed-Int> ")" ;
<Signed-Int>   ::= +Integer | -Integer | Integer ;
```

An LL(1) Derivation.

Consider finding the derivation sequence for the sentence

(1,-2) home end

We start with the distinguished symbol and its 1 production

<Session>

→ **<Instruction> <More> end**

The next symbol '(' tells us to use the 3rd production

→ **<Coord> <More> end**

→ **(<Signed-Int>, <Signed-Int>) <More> end**

We now have the '(' and the next symbol is an integer

so use the 3rd production of <Signed-Int>

→ **(<Integer>, <signed-Int>) <More> end**

→ **(1, <Signed-Int>) <More> end**

Completing the Derivation.

(1, <Signed-Int>) <More> end

We now have '(1,' and the next symbol is '-' which gives

→ **(1, -<Integer>) <More> end**

→ **(1, -2) <More> end**

We now have '(1, -2)' and the next symbol is 'home' giving

→ **(1, -2) <Instruction> <More> end**

→ **(1, -2) home <More> end**

The next input symbol is now 'end'.

This tells us to use the null production for <More> giving

→ **(1, -2) home end**

This completes the derivation sequence. At every step, we could choose one of the productions just based on the next symbol.

Which production to choose?

- Each production in a rule is associated with a set of terminal symbols that, if any occur as the next input symbol, indicate that the corresponding production of the rule should be chosen for the next derivation step.
- These sets are known as **director sets**.

The LL(1) Condition.

- The **LL(1) condition** can be stated thus:
- ***A BNF specification is LL(1) if, for every rule, the director sets for the productions of that rule are disjoint.***
- That is, there must be no common terminal symbols (**ambiguity**) in the director sets of the productions of any rule.

Example of Director Sets.

- Consider the BNF:

  <X> ::= <Y> | <Z> | e f ;
  <Y> ::= a <Z> | b ;
  <Z> ::= c | d ;

- The director sets for the productions of <X>
  - <X> ::= <Y> **{ "a", "b" }**
  - <X> ::= <Z> **{ "c", "d" }**
  - <X> ::= e f **{ "e" }**

Another Example.

- Here are the director sets for the <Session> BNF specification provided earlier.
- <More> ::= <Instruction> <More>
  {"home", "forward", "back", "left", "right","(" }
- <More> ::= <>
  { "end" }
- <Instruction> ::= home
  { "home" }
- <Instruction> ::= <Direction> <Distance>
  { "forward", "back", "left", "right" }
- <Instruction> ::= <Coord>
  { "(" }

Another Example Continued.

- <Distance> ::= Integer
  { *any integer* }
- <Distance> ::= <>
  {"home","forward","back","left","right","(", "end"}
- <Signed-int> ::= + Integer
  { "+" }
- <Signed-Int> ::= - Integer
  { "-" }
- <Signed-Int> ::= Integer
  { *any integer* }

- The director sets for <Direction> should be obvious.

LL(1) Specifications.

- We will primarily be using LL(1) specifications in this module.
- We have outlined how a BNF can be checked for this property by working out the director sets for the productions of each rule.
- In practice this can usually be done by inspection – but be careful with <> – you must look at the whole language to find the productions that might follow it!

Avoiding LL(1) Conflicts

- If a BNF specification is not LL(1) then certain transformations can be applied to avoid the non-LL(1) conflicts.
- Factorisation allows common starter symbols to be removed.
- Removing direct Left Recursion may be accomplished through transforming the relevant rule.
- I'm going to go through this fairly quickly for now, but please **refer back** to these slides when you need to!

Factorisation

- Rules can be factored on any common component across productions.

  <A> ::= α β | α δ | ... . →→ <A> ::= α <X> | ... .
  <X> ::= β | δ .

- Using EBNF this can be expressed as

  <A> ::= α β | α δ | ... . →→ <A> ::= α (β | δ) | ... .

Rearrangement

- Rules may need rearranging before factorisation:

<Stat> ::= <Assign> | <Call> .
<Assign> ::= <ident> = <Expr> .
<Call> ::= <ident> (<Params>) .

Rule is not LL(1):
both productions start with <ident>.

- Rearrange and then factorise:

<Stat> ::= <ident> [= <Expr> | (<Params>)] .

Recursive Rules …

- are those in which at least one production of the rule contains the target non-terminal.
- are extremely common since recursion is the basic construct for introducing repetition.

- come in one of three flavours of productions
  - *left recursive* <A> ::= <A> α .
  - *right recursive* *<A> ::=* α *<A>.*
  - *central recursive* *<A> ::=* α *<A>* β*.*

Left Recursion

- Left recursive productions are undesirable for LL(1) grammars; any rule that has a left-recursive production cannot be LL(1).
  - e.g. **<A> ::= <A> α | β .**
- The director set for the 1st production (<A> α) must contain all the symbols in the director set for the 2nd production (β) since the <A> can derive the β.
  - Thus the director sets are not disjoint, and
  - hence the rule is not LL(1).

Left Recursion Example

- Consider the rule
  <List> ::= <List> , i | i .
- and we wish to find the derivation **sequence** for the input string
  i , i , i
- We start with
  <List> → ?
- the next input symbol is 'i' but we cannot tell if this is for the 1st or 2nd production of the rule for <List>.

Types of Left Recursion

- Left recursion can arise in two ways:
  - direct left recursion, e.g.
    <A> ::= <A> a | b .
  - Indirect left recursion
    <A> ::= <B> | a .
    <B> ::= <C> | b .
    <C> ::= <A> | c .

- Indirect left recursion can be difficult to identify and to remove.

Removing Direct Left Recursion

- Direct left recursion can be removed using the following general transformation

<A> ::= <A> α | β | δ | ... . →→ **<A> ::= <X> <Y> .**
<X> ::= β | δ ... .
<Y> ::= α <Y> | <> .

- For a simpler rule this can be expressed as

<A> ::= <A> α | β . →→ **<A> ::= β <Y> .**
<Y> ::= α <Y> | <> .

Removing Direct Left Recursion With EBNF

- With EBNF the transformation becomes

  <A> ::= <A> α | β | δ | ... .

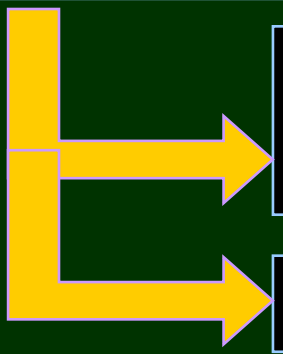
  ➔➔ <A> ::= (β | δ ...) { α }* .

- For our simpler rule this can be expressed as

  <A> ::= <A> α | β . **➔➔ <A> ::= β (α)* .**

Removal Examples

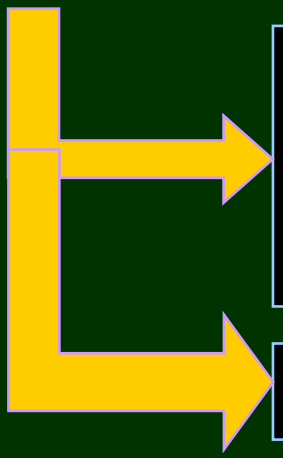
<List> ::= <List> , <Item> | <Item> .



<List> ::= <Item> <Rest> .
<Rest> ::= , <Item> <Rest> | <> .

<List> ::= <Item> { , <Item> }* .

<Funny> ::= <Funny> :-) | ha ha | <Name> ! .



<Funny> ::= <Start> <Rest> .
<Start> ::= ha ha | <Name> ! .
<Rest> ::= :-) <Rest> | <> .

<Funny> ::= [ha ha | <Name> !] { :-) }* .

Microsyntax

- Motivation.
- Layout and Whitespace.
- Microsyntax.
- Elements of Microsyntax.

Microsyntax.

- Before we parse our language, we first need to break it up into **tokens** (terminal symbols).
- To do this, we need a specification of the **lexical structure** of our language.
- For the purposes of this module, Allan's EBNF toolkit lets you embed this in the EBNF specification – as “microsyntax”.

- (We'll look next week at how we actually do tokenisation in practice.)

An Example Using Microsyntax.

- <Expression> ::= <Value> (+|-) <Value> ;
  <Value> ::= Identifier | Integer ;

- microsyntax // very informal
  Identifier <|as usual
  Integer <|one or more digits

- microsyntax // formal (what we'll normally do)
  Identifier <|[a-zA-Z]\w*
  Integer <|\d+

Defining microsyntax in EBNF

- microsyntax
  Identifier <|[a-zA-Z]\w*
  Integer <|\d+

Name of the terminal

This is just punctuation...

A regular expression (in .NET syntax)

- See Allan's “EBNF v3” spec for more details

Defining Lexical Structure

- Nothing says that every language must have the same definition of the microsyntax entities.
  - The lexical structure can be defined formally or informally for any specific language.
  - This definition can include as much or as little detail as appropriate.
- Nothing in the lexical structure affects the main BNF/EBNF.
  - In our EBNF, the microsyntax names are treated as terminal symbols in the BNF proper. But you could change the microsyntax if, say, you wanted to allow new characters in identifiers...

Identifiers.

- We saw a typical lexical rule for an identifier.
  - It distinguished between what could start the identifier and what could be within it.
- Sometimes other characters such as periods or underscores (or even spaces!) are allowed either to start or be part of an identifier.
- For example

microsyntax
Identifier <|[a-zA-Z][\w._]*

Literals.

- Literal constants are also common in languages.
  - Integers, real numbers, booleans, characters, strings.
- These also are usually identified by the lexer, and treated as terminals in the main BNF.
- It is usually better if unary numeric signs (+,-) are handled in the main BNF.
- String constants get particularly complicated!
  - Break across multiple lines?
    Escape characters? Embedded quotes?

Any questions?

- Coursework available from MyLearningSpace – please have a look at this and ask if you've got any questions

- **Next lecture**: lexical analysis