

# An Introduction to Parsing

**COMP2600 — Formal Methods for Software Engineering**

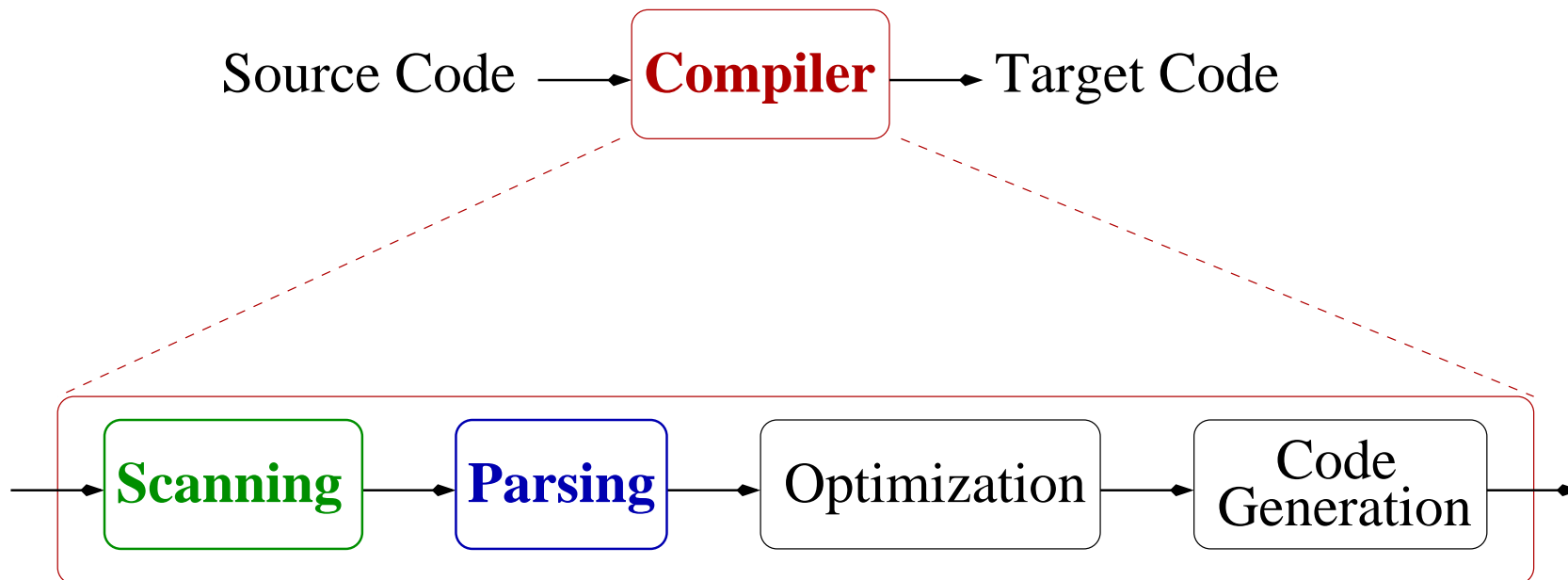
Jeremy Dawson

Australian National University

Semester 2, 2011

## Components of a Compiler

A **compiler** is a program that translates programs in a *source language* to equivalent programs in a *target language* (usually a low-level language such as assembler, machine-code or the language of an abstract machine such as the JVM).



# Scanning

Also known as **lexical analysis**.

Converts the source *text* into a sequence of *tokens*: identifiers, literal numerals, keywords, punctuation, etc.

**Regular languages** are used to describe tokens — they are about the right power.

**Scanners** are based on *FSAs* — either generated or hand-coded.

# Parsing

Also known as **syntax analysis**. The sequence of tokens from the scanner is analysed to retrieve the “*structure*” of the program — e.g the body of a loop, the parameters of a function declaration.

**Context-free languages** are used to describe the syntax of programming languages – they are about the right power.

(CFGs can also describe some *semantic* properties such as operator precedence. On the other hand, there are some *syntactic properties*, such as requiring all variables that are used to have been declared, that CFGs **cannot** represent.)

**Parsers** are based on *PDA*s — either generated or hand-coded.

## Top-Down Parsing

Most (but certainly not all) parser *generators* take a *bottom-up* approach, but top-down is easier to grasp. *Hand-coded* parsers are usually *top-down*.

The basic idea is that the *parse tree* is built from the *root* down to the *leaves*.

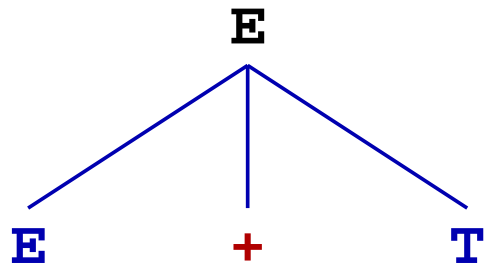
### Example:

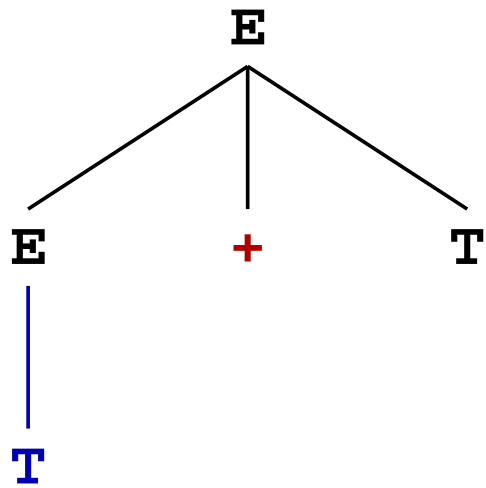
$$E \rightarrow E + T \mid E - T \mid T$$

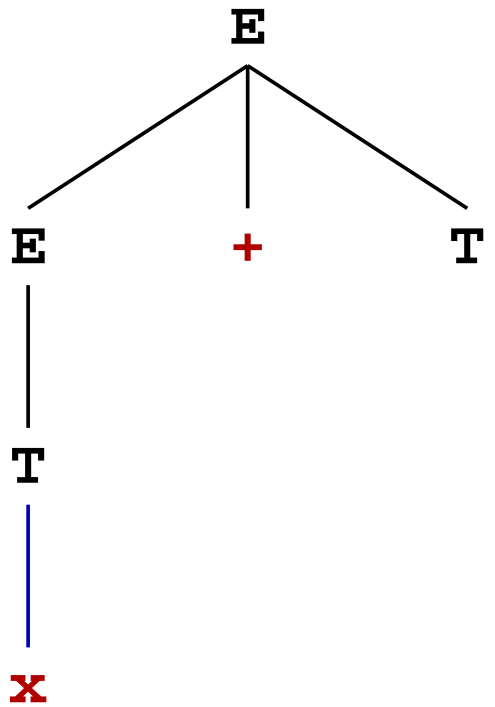
$$T \rightarrow \mathbf{id} \mid (E)$$

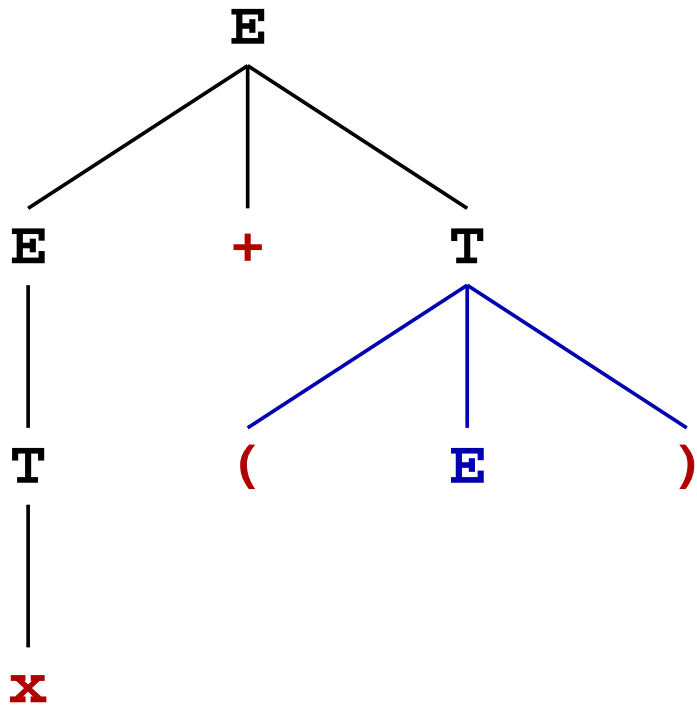
Build the parse tree for

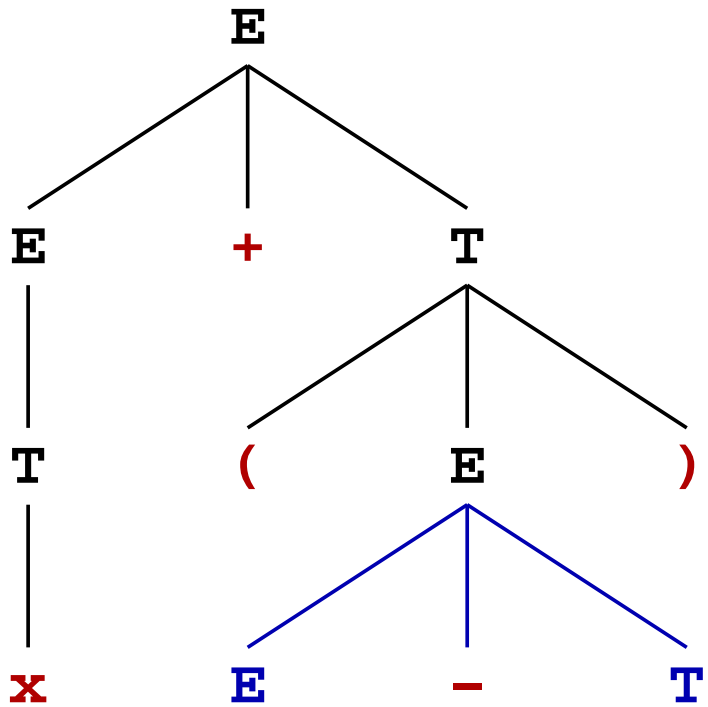
$$x + (y - z)$$

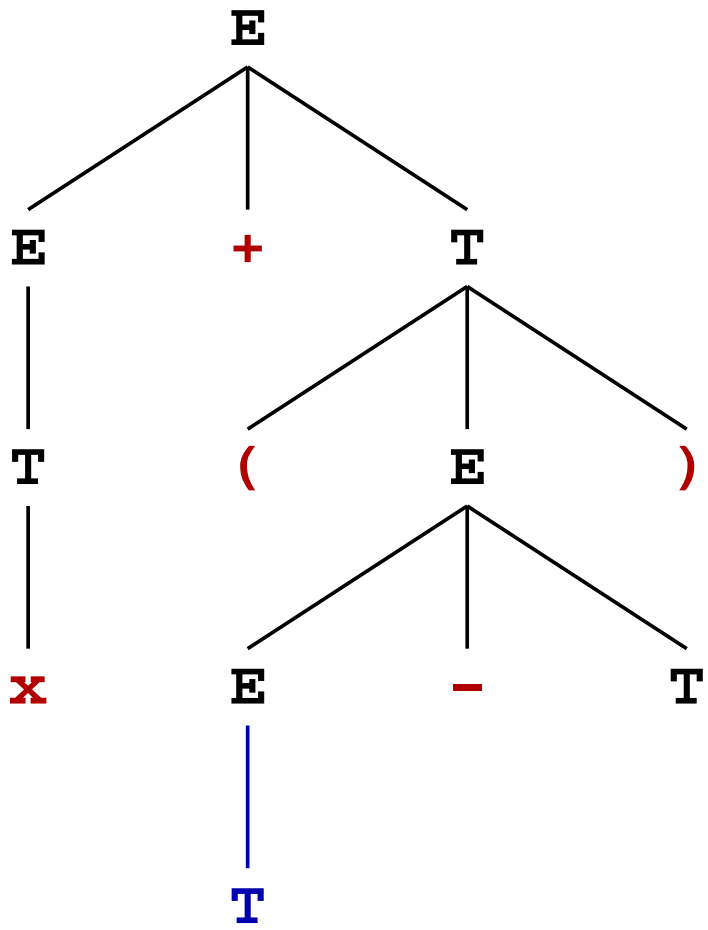




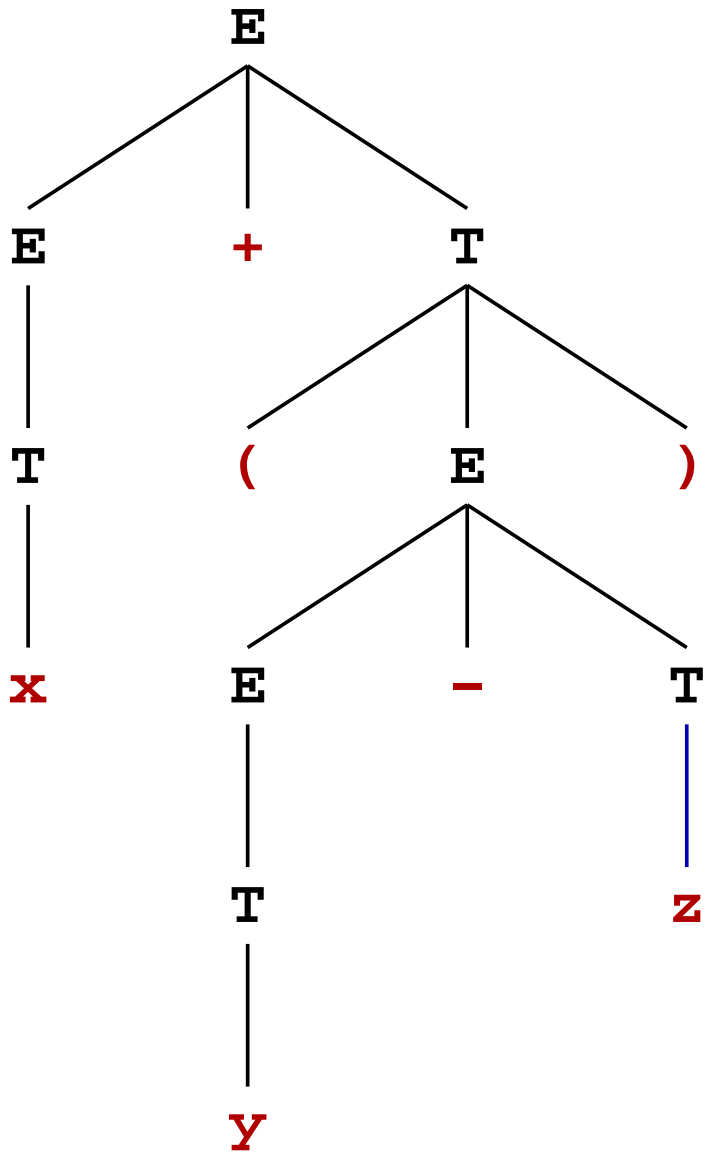












## CFG to PDA revisited

The following CFG is for a simple language of type expressions:

$$\begin{aligned} type &\rightarrow simple \mid *id \mid \mathbf{array} [simple] \mathbf{of} type \\ simple &\rightarrow \mathbf{integer} \mid \mathbf{char} \mid \mathbf{num} \text{ ellipsis } \mathbf{num} \end{aligned}$$

(Assume we have a scanner to recognise all the tokens.)

We know how to derive a *non-deterministic PDA* from this grammar...

(Only the production expansions are interesting. We leave out the transitions which consume an input symbol and pop a matching symbol from the stack. We also leave out the state which is  $q_1$  everywhere.)

## PDA for the *types* Grammar

$type \rightarrow simple \mid *id \mid \mathbf{array} [simple] \mathbf{of} type$   
 $simple \rightarrow \mathbf{integer} \mid \mathbf{char} \mid \mathbf{num} \text{ ellipsis } \mathbf{num}$

$\delta(\varepsilon, type) \mapsto simple$

$\delta(\varepsilon, type) \mapsto *id$

$\delta(\varepsilon, type) \mapsto \mathbf{array} [simple] \mathbf{of} type$

$\delta(\varepsilon, simple) \mapsto \mathbf{integer}$

$\delta(\varepsilon, simple) \mapsto \mathbf{char}$

$\delta(\varepsilon, simple) \mapsto \mathbf{num} \text{ ellipsis } \mathbf{num}$

**Notice that the choice of production is made without considering the next token in the stream.**

## Looking Ahead

Can we make the PDA deterministic by using the next token in the stream (the **lookahead symbol**) to help choose which production?

For this grammar, clearly yes. These are obvious:

$$\begin{aligned}\delta(\varepsilon[*], type) &= *id \\ \delta(\varepsilon[\mathbf{array}], type) &= \mathbf{array} [simple] \mathbf{of} type \\ \delta(\varepsilon[\mathbf{integer}], simple) &= \mathbf{integer} \\ \delta(\varepsilon[\mathbf{char}], simple) &= \mathbf{char} \\ \delta(\varepsilon[\mathbf{num}], simple) &= \mathbf{num} \mathbf{elipsis} \mathbf{num}\end{aligned}$$

**NOTE:** The notation  $\varepsilon[x]$  is used to indicate that for these transitions we **look at**, but **don't** consume, the input symbol  $x$ .

That is, if *simple* is on top of stack and **num** is the next token on input, replace *simple* with the sequence **num elipsis num**.

## Looking Ahead ctd

If there is a *type* symbol on top of stack, then any token that can start a *simple* tells us to choose the *type*  $\rightarrow$  *simple* production:

$$\delta(\varepsilon[\mathbf{integer}], \textit{type}) = \textit{simple}$$

$$\delta(\varepsilon[\mathbf{char}], \textit{type}) = \textit{simple}$$

$$\delta(\varepsilon[\mathbf{num}], \textit{type}) = \textit{simple}$$

Again, the notation  $\varepsilon[x]$  means we **look at**, but **don't** consume input symbol  $x$ . The PDA will also have transitions which, when the input symbol is also on top of the stack, consume the input symbol and pop the top stack symbol.

If a lookahead symbol doesn't match the top of stack non-terminal symbol as specified in the cases above, the input string is not a sentence of the language and will be rejected.

## Recursive Descent Parsers

The common approach to *hand-coding* parser is called **recursive descent** and uses the lookahead symbol in an analogous way to the PDA example above. A recursive descent parser consists of a collection of functions, one associated with *each non-terminal* of the grammar.

Within each of those functions, depending on the lookahead symbol (i.e. the next token in the input stream), either a token is checked and skipped or a (non-terminal) function is called, in sequence according to the rhs of each production.

A Haskell implementation of such a parser is available on the course web site: [ParseTypeDefs.hs](#)

## A General Solution?

Using the lookahead symbol to direct the parse choices looks easy enough but the *types* grammar was carefully chosen. This approach only works for a class of CFGs known as **LL(1)** grammars.

(The first **L** stands for “left-to-right scan of the input; the second **L** stands for “leftmost parse” and the **1** indicates “one symbol lookahead.”)

There are standard ways (algorithms) of massaging CFGs into LL(1) form, if possible.

Let's look at some of the main problems and how they may be overcome. . .

## Another Example

Suppose we were trying to make the PDA for the expression grammar deterministic, by using the lookahead symbol.

$$E \rightarrow T \mid E + T \mid E - T$$

$$T \rightarrow F \mid T * F$$

$$F \rightarrow \mathbf{id} \mid (E)$$

If there is a  $F$  on top of stack, then clearly the only valid next tokens are **id** and  $($  and the PDA transitions are obvious:

$$\delta(\varepsilon[\mathbf{id}], F) \mapsto \mathbf{id}$$

$$\delta(\varepsilon[(], F) \mapsto (E)$$

## Example ctd

But what if  $T$  is on top of stack and **id** is the lookahead. That would be compatible with both productions

$$T \rightarrow F$$

$$T \rightarrow T * F$$

So both these PDA transitions would be valid

$$\delta(\varepsilon[\mathbf{id}], T) \mapsto F$$

$$\delta(\varepsilon[\mathbf{id}], T) \mapsto T * F$$

which is not deterministic.

To correctly choose between these two productions would require *double lookahead* to see if there is a  $*$  coming.

## Example ctd

But what if  $E$  is on top of stack and **id** is the lookahead. That would be compatible with these productions

$$E \rightarrow T$$

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

To correctly choose between these productions would require an *unbounded lookahead* to see if there is a  $+$  or a  $-$  coming.

## Left Recursive Productions

The problem above occurs whenever there are **left recursive** productions:

$$A \rightarrow A\dots$$

There is a general algorithm for eliminating left recursion (for non-ambiguous grammars). The grammar may become less natural but that's the price we pay ...

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow \mathbf{id} \mid (E)$$

## First and Follow

If the lookahead symbol can *start* the right hand side of a production, choose that production. For *empty* productions the lookahead must be a token that can *follow* that non-terminal. ( $\dagger$  represents end of input):

$E \rightarrow TE'$  on lookahead: **id** (

$E' \rightarrow \varepsilon$  on lookahead: )  $\dagger$

$E' \rightarrow +TE'$  on lookahead: +

$T \rightarrow FT'$  on lookahead: **id** (

$T' \rightarrow \varepsilon$  on lookahead: + )  $\dagger$

$T' \rightarrow *FT'$  on lookahead: \*

$F \rightarrow \mathbf{id}$  on lookahead: **id**

$F \rightarrow (E)$  on lookahead: (

A simple Haskell implementation of a scanner, parser and evaluator for simple arithmetic expressions is on the lectures page of the course web site:

`AbSyn.hs`, `Scanner.hs`, `Parser.hs`, `Evaluator.hs`, `Main.hs`