Introduction to Compiler Construction

Parsing: Preliminaries

Outline

Parser

Parser

The process of parsing a program is to determine its syntactic structure

Parser

The process of parsing a program is to determine its syntactic structure

The program that does this is called a parser

Parser

The process of parsing a program is to determine its syntactic structure

The program that does this is called a parser

A parser should:

- Make sure the program is syntactically valid, ie, conforms to the grammar describing the program's syntax
- Identify syntax errors and report them along with the line numbers they appear on
- Not stop on the first error, but report the error, recover from it, and look for additional errors
- Produce an intermediate representation (IR) of the parsed program that is suitable for semantic analysis

Parser

The process of parsing a program is to determine its syntactic structure

The program that does this is called a parser

A parser should:

- Make sure the program is syntactically valid, ie, conforms to the grammar describing the program's syntax
- Identify syntax errors and report them along with the line numbers they appear on
- Not stop on the first error, but report the error, recover from it, and look for additional errors
- Produce an intermediate representation (IR) of the parsed program that is suitable for semantic analysis

In *j--*, the IR is in the form of an abstract syntax tree (AST)

Parser

Example: parsing the *j*-- program

</> HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

results in the following AST

Parser

Parser

```
{
    "JCompilationUnit:3":
    {
        "source": "tests/HelloWorld.java", "imports": ["java.lang.System"],
        "JClassDeclaration:5":
        {
            "modifiers": ["public"],
            "name": "HelloWorld",
            "super": "java.lang.Object",
            "JMethodDeclaration:7":
            {
                "modifiers": ["public", "static"],
                "returnType": "void",
                "name": "main",
                "parameters": [["args", "String[]"]],
                "JBlock:7":
                {
                    "JStatementExpression:8":
                    {
                        "JMessageExpression:8":
                        {
                            "ambiguousPart": "System.out", "name": "println",
                            "Argument":
                            {
                                "JLiteralString:8":
                                {
                                    "type": "", "value": "Hello, World"
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Parser

Parser

The nodes in the AST represent syntactic objects and the edges correspond to their attributes

Parser

The nodes in the AST represent syntactic objects and the edges correspond to their attributes

The tree representation for a program is easier to analyze and decorate (with type information) than text

Parser

The nodes in the AST represent syntactic objects and the edges correspond to their attributes

The tree representation for a program is easier to analyze and decorate (with type information) than text

The AST makes the syntax implicit in the program text, explicit

Context-free Grammars and Languages

Context-free Grammars and Languages

Recursive programming languages such as *j*-- are best described by context-free grammar rules, using a notation called Backus-Naur Form (BNF)

Recursive programming languages such as *j--* are best described by context-free grammar rules, using a notation called Backus-Naur Form (BNF)

Example: the rule

$$S ::= \texttt{if (}E\texttt{)}\ S$$

says that, if E is an expression and S is a statement, then

$$\texttt{if (}E\texttt{)}\ S$$

is also a statement

Context-free Grammars and Languages

Example: the rule

$$S ::= \texttt{if (}E\texttt{)}\ S \\ \quad \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S$$

is shorthand for

$$S ::= \texttt{if (}E\texttt{)}\ S$$
$$S ::= \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S$$

Context-free Grammars and Languages

Example: the rule

$$S ::= \texttt{if (}E\texttt{)}\ S$$
$$\quad\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S$$

is shorthand for

$$S ::= \texttt{if (}E\texttt{)}\ S$$
$$S ::= \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S$$

Square brackets indicate that a phrase is optional

Example: the rule

$$S ::= \texttt{if (}E\texttt{)}\ S$$
$$\quad |\ \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S$$

is shorthand for

$$S ::= \texttt{if (}E\texttt{)}\ S$$
$$S ::= \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S$$

Square brackets indicate that a phrase is optional

Example: the two rules from above can be written as

$$S ::= \texttt{if (}E\texttt{)}\ S\ [\texttt{else}\ S]$$

Context-free Grammars and Languages

Curly braces denote the Kleene closure, indicating that the phrase may appear zero or more times

Curly braces denote the Kleene closure, indicating that the phrase may appear zero or more times

Example: the rule

$$E ::= T \{ + \ T \}$$

says that an E may be written as T (for term) followed by zero or more occurrences of + followed by T, such as

$$T + T + T + T \ldots$$

Context-free Grammars and Languages

One may use the alternation sign | to denote a choice and parentheses for grouping

One may use the alternation sign | to denote a choice and parentheses for grouping

Example: the rule

$$E ::= T \; \{(\texttt{+} \mid \texttt{-}) \; T\}$$

says that the additive operator may be either + or -, such as

$$T \; \texttt{+} \; T \; \texttt{-} \; T \; \texttt{+} \; T \; \ldots$$

Context-free Grammars and Languages

Example (BNF rules in *j--*):

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }

typeDeclaration ::= modifiers classDeclaration

modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }

classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody

classBody ::= LCURLY { modifiers memberDecl } RCURLY
```

Context-free Grammars and Languages

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals
- T is a set of terminals
- $S \in N$ is the start symbol
- P is a set of productions (aka rules)

Context-free Grammars and Languages

Context-free Grammars and Languages

Example: arithmetic expression grammar $A = (N, T, S, P)$, where

- $N = \{E, T, F\}$
- $T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$
- $S = E$
- P is the following set of productions

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } F \\
T &::= F \\
F &::= \texttt{(}E\texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Example: arithmetic expression grammar $A = (N, T, S, P)$, where

- $N = \{E, T, F\}$
- $T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$
- $S = E$
- P is the following set of productions

$$E ::= E \texttt{ + } T$$
$$E ::= T$$
$$T ::= T \texttt{ * } F$$
$$T ::= F$$
$$F ::= \texttt{(}E\texttt{)}$$
$$F ::= \texttt{id}$$

A grammar can be specified informally as a sequence of productions

Context-free Grammars and Languages

From the start symbol, using productions, we can generate infinitely many strings

Example (strings generated from the start symbol E in A):

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow \texttt{id} + T \\
&\Rightarrow \texttt{id} + T * F \\
&\Rightarrow \texttt{id} + F * F \\
&\Rightarrow \texttt{id} + \texttt{id} * F \\
&\Rightarrow \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

Context-free Grammars and Languages

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string

Example (derivations using productions in A):

$$E \overset{*}{\Rightarrow} E \text{ (in zero steps)}$$
$$E \overset{*}{\Rightarrow} \texttt{id + } F \texttt{ * } F$$
$$T \texttt{ + } T \overset{*}{\Rightarrow} \texttt{id + id * id}$$

Context-free Grammars and Languages

The language $L(G)$ described by a grammar G consists of all the strings comprised of only terminal symbols, ie,
$L(G) = \{w | S \overset{*}{\Rightarrow} w \text{ and } w \in T*\}$

The language $L(G)$ described by a grammar G consists of all the strings comprised of only terminal symbols, ie, $L(G) = \{w | S \stackrel{*}{\Rightarrow} w \text{ and } w \in T*\}$

Example (the language $L(A)$):

$$E \stackrel{*}{\Rightarrow} \texttt{id}$$
$$E \stackrel{*}{\Rightarrow} \texttt{id + id * id}$$
$$E \stackrel{*}{\Rightarrow} \texttt{(id + id) * id}$$

so, $L(A)$ includes each of

```
id
id + id * id
(id + id) * id
```

and infinitely more finite strings

Context-free Grammars and Languages

A left-most derivation is a derivation in which at each step, the next string is derived by applying a production for rewriting the left-most non-terminal

Example (left-most derivation using productions in A):

$$\begin{aligned} \underline{E} &\Rightarrow \underline{E} + T \\ &\Rightarrow \underline{T} + T \\ &\Rightarrow \underline{F} + T \\ &\Rightarrow \texttt{id} + \underline{T} \\ &\Rightarrow \texttt{id} + \underline{T} * F \\ &\Rightarrow \texttt{id} + \underline{F} * F \\ &\Rightarrow \texttt{id} + \texttt{id} * \underline{F} \\ &\Rightarrow \texttt{id} + \texttt{id} * \texttt{id} \end{aligned}$$

Context-free Grammars and Languages

A right-most derivation is a derivation in which at each step, the next string is derived by applying a production for rewriting the right-most non-terminal

Example (right-most derivation using productions in A):

$$
\begin{aligned}
\underline{E} &\Rightarrow E + \underline{T} \\
&\Rightarrow E + T * \underline{F} \\
&\Rightarrow E + \underline{T} * \texttt{id} \\
&\Rightarrow E + \underline{F} * \texttt{id} \\
&\Rightarrow \underline{E} + \texttt{id} * \texttt{id} \\
&\Rightarrow \underline{T} + \texttt{id} * \texttt{id} \\
&\Rightarrow \underline{F} + \texttt{id} * \texttt{id} \\
&\Rightarrow \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

Context-free Grammars and Languages

A sentential form is any string of terminal and non-terminal symbols that can be derived from the start symbol

Context-free Grammars and Languages

A sentential form is any string of terminal and non-terminal symbols that can be derived from the start symbol

A sentence is any string of only terminal symbols that can be derived from the start symbol

A sentential form is any string of terminal and non-terminal symbols that can be derived from the start symbol

A sentence is any string of only terminal symbols that can be derived from the start symbol

Example (sentential forms and sentences derived using the start symbol E in A):

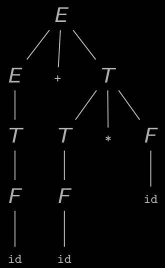
E
$E + T$
$E + T * F$
...
F `+ id * id`
`id + id * id`

are all sentential forms and `id + id * id` is a sentence

Context-free Grammars and Languages

A parse tree illustrates the derivation (and structure) of a sentence (at the leaves) from a start symbol (at the root)

Example (parse tree for `id + id * id` in $L(A)$):



Ambiguous Grammars and Languages

Ambiguous Grammars and Languages

Given a grammar G, if there exists a sentence $s \in L(G)$ for which there are more than one left- (or right-) most derivations or parse trees, we say the sentence s is ambiguous

Ambiguous Grammars and Languages

Given a grammar G, if there exists a sentence $s \in L(G)$ for which there are more than one left- (or right-) most derivations or parse trees, we say the sentence s is ambiguous

If a grammar G derives at least one ambiguous sentence, we say the grammar G is ambiguous; if there is no such sentence, we say the grammar is unambiguous

Ambiguous Grammars and Languages

Ambiguous Grammars and Languages

Example (ambiguous arithmetic expression grammar A_{amb}):

$$E ::= E \texttt{+} E \mid E \texttt{*} E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

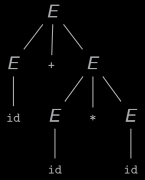
Ambiguous Grammars and Languages

Example (ambiguous arithmetic expression grammar A_{amb}):

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

One left-most derivation and corresponding parse tree for the sentence $\texttt{id + id * id}$:

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ + } E \\
&\Rightarrow \texttt{id + } \underline{E} \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$



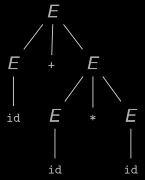
Ambiguous Grammars and Languages

Example (ambiguous arithmetic expression grammar A_{amb}):

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

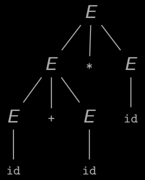
One left-most derivation and corresponding parse tree for the sentence `id + id * id`:

$$\begin{aligned} \underline{E} &\Rightarrow \underline{E} \texttt{ + } E \\ &\Rightarrow \texttt{id + } \underline{E} \\ &\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\ &\Rightarrow \texttt{id + id * } \underline{E} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$



Another left-most derivation and corresponding parse tree for `id + id * id`:

$$\begin{aligned} \underline{E} &\Rightarrow \underline{E} \texttt{ * } E \\ &\Rightarrow \underline{E} \texttt{ + } E \texttt{ * } E \\ &\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\ &\Rightarrow \texttt{id + id * } \underline{E} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$



Ambiguous Grammars and Languages

Ambiguous Grammars and Languages

Example (dangling-else problem):

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
  &\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
  &\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

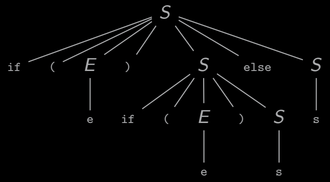
Ambiguous Grammars and Languages

Example (dangling-else problem):

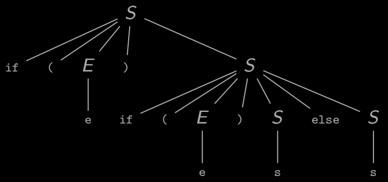
$$
\begin{aligned}
S \;::=\;& \texttt{if (}E\texttt{)}\ S \\
\mid\;& \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
\mid\;& \texttt{s} \\
E \;::=\;& \texttt{e}
\end{aligned}
$$

Two left-most derivations and corresponding parse trees for the sentence `if (e) if (e) s else s`

$$
\begin{aligned}
\underline{S} &\Rightarrow \texttt{if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e) s else}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (e) s else s}
\end{aligned}
$$



$$
\begin{aligned}
\underline{S} &\Rightarrow \texttt{if (}\underline{E}\texttt{)}\ S \\
&\Rightarrow \texttt{if (e)}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e) s else}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (e) s else s}
\end{aligned}
$$



Ambiguous Grammars and Languages

Ambiguous Grammars and Languages

Resolution of the dangling-else problem:

$$
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

Ambiguous Grammars and Languages

Resolution of the dangling-else problem:

$$
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\ \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\ \mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

But programmers have become accustomed to (and fond of) the ambiguous conditional

Ambiguous Grammars and Languages

Resolution of the dangling-else problem:

$$
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

But programmers have become accustomed to (and fond of) the ambiguous conditional

Compiler writers handle the rule as a special case in the parser such that an `else` is grouped along with the closest preceding `if`

Ambiguous Grammars and Languages

Ambiguous Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w` (eg, `java.lang.Math.PI`)

Ambiguous Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w` (eg, `java.lang.Math.PI`)

Clearly `w` is a field, but what about `x.y.z`?

Ambiguous Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w` (eg, `java.lang.Math.PI`)

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

Ambiguous Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w` (eg, `java.lang.Math.PI`)

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

`x.y` might be a package in which the class `z` is defined, and `w` a static field in that class

Ambiguous Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w` (eg, `java.lang.Math.PI`)

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

`x.y` might be a package in which the class `z` is defined, and `w` a static field in that class

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until semantic analysis

Ambiguous Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w` (eg, `java.lang.Math.PI`)

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

`x.y` might be a package in which the class `z` is defined, and `w` a static field in that class

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until semantic analysis

The parser represents `x.y.z` in the AST as an `AmbiguousName` node, which gets reclassified during semantic analysis