CS2 Language Processing note 12

Automatic generation of parsers

In this note we describe how one may automatically generate a parse table from a given grammar (assuming the grammar is $LL(1)$). This and similar methods can be used in practice to build *parser generators* — programs which take a grammar supplied by the user and automatically write the code for a parser for the corresponding languages. We end the note with a brief discussion of lexer and parser generators.

As we have seen, for small examples of grammars it is often possible to work out a parse table by hand simply by inspection of the grammar. For a large grammar, however, this is no longer feasible, and we require a more systematic method for computing the parse table. The method we shall describe here also has the virtue that it can be carried out completely automatically, and hence one can write a program that computes parse tables for us.

First and *Follow* sets

To generate a parse table from a given grammar, we need some preliminary data about which tokens a parser might meet when considering any particular nonterminal. This takes the form of two sets for each nonterminal A, known as *First*(A) and *Follow*(A). One can think of these as follows: Suppose at some point during parsing, we are expecting a phrase corresponding to some nonterminal A, and we see a terminal a as the next symbol in the input. Assuming the sentence we are parsing is a valid one, there are two possible kinds of reasons for the appearance of this a: either it is the first symbol of the A-phrase we are expecting, or this A-phrase is in fact the empty string and the a we see is the start of whatever comes after it. We are therefore interested in both "what an A can possibly start with" and "what an A can possibly be followed by in a complete sentence". More precisely:

First: For any nonterminal *A*, the set *First*(*A*) comprises all terminals that can appear at the start of some sentential form derived from *A*. In addition, if *A* derives the empty string, then *First*(*A*) includes the symbol ε.

Follow: For any nonterminal *A*, the set *Follow*(*A*) is made up of all the terminals that can appear after *A* in any sentential form derived from some nonterminal. In addition, *Follow*(*A*) contains the token $ if *A* can appear at the end of some sentential form derived from the start symbol for the grammar.

For those who like a more mathematical notation:

$$\mathit{First}(A) = \{\, a \mid \exists u \,.\, A \Rightarrow^* au \,\} \cup \{\, \varepsilon \mid A \Rightarrow^* \varepsilon \,\}$$

$$\mathit{Follow}(A) = \{\, a \mid \exists B,u,v \,.\, B \Rightarrow^* uAav \,\} \cup \{\, \$ \mid \exists u \,.\, S \Rightarrow^* uA \,\}$$

Computing *First* sets

We now give a systematic method for computing *First* and *Follow* sets. We will illustrate our method by applying it to the following grammar G. This describes a language of commands such as one might type to a shell, where a command name like `cp` or `ls` is followed by some options and some file names.

shell	$\longrightarrow$	command *args*
args	$\longrightarrow$	*opts files*
opts	$\longrightarrow$	option *opts* $\mid \varepsilon$
files	$\longrightarrow$	file *files* $\mid \varepsilon$

As a first stage, we construct the set E of *potentially empty* nonterminals, i.e. those nonterminals A such that $A \Rightarrow^* \varepsilon$. The general method is as follows:

1. Start by setting E to be the set of nonterminals A such that the grammar contains a production $A \longrightarrow \varepsilon$.
2. For every production $A \longrightarrow t$ in the grammar, if t consists entirely of nonterminals in E, add A itself to E.
3. Repeat step 2 until E stabilizes.

Applying this method to the grammar G, we see at step 1 that *opts*, *files* $\in E$. At step 2 we then note that since we have a production *args* $\longrightarrow$ *opts files*, we must also have *args* $\in E$. Since further applications of step 2 do not yield any new elements of E, we conclude that $E = \{\mathit{opts}, \mathit{files}, \mathit{args}\}$.

We now give the algorithm for computing *First* sets. In fact, it will be convenient to compute a set $\mathit{First}(x)$ for *every* symbol x, terminal or nonterminal.

1. Start by setting $\mathit{First}(a) = \{a\}$ for each terminal a; $\mathit{First}(A) = \{\varepsilon\}$ for each nonterminal $A \in E$; and $\mathit{First}(A) = \emptyset$ for each nonterminal $A \notin E$.
2. For each production $A \longrightarrow x_1 x_2 \cdots x_n$ (where the x_i's may be terminals or nonterminals), and for each i ($1 \leq i \leq n$) such that $x_j \in E$ whenever $1 \leq j \leq i-1$, add every terminal $a \in \mathit{First}(x_i)$ to $\mathit{First}(A)$.
3. Repeat step 2 until all the sets $\mathit{First}(A)$ stabilize.

Let us apply this algorithm to our grammar G. We start by setting $\mathit{First}(a) = \{a\}$ for each of the terminals $a =$ **command**, **option**, **file**; $\mathit{First}(A) = \{\varepsilon\}$ for $A =$

args, *opts*, *files*; and *First*(*shell*) $= \emptyset$. For step 2, we need to consider in turn each production and each possible value of i for that production. For the production *shell* $\rightarrow$ command *args*, taking $i = 1$ gives us that command $\in$ *First*(*shell*), and we need not consider $i = 2$ since command $\notin E$. The production *args* $\rightarrow$ *opts files* yields nothing at the moment, since *First*(*opts*) and *First*(*files*) do not at present contain any terminals. The remaining productions, however, yield that option $\in$ *First*(*opts*) and file $\in$ *First*(*files*). This completes the first "round" of step 2. On the second round, however, we discover something new when we consider the rule *args* $\rightarrow$ *opts files*: by taking $i = 1, 2$ respectively, we find that option, file $\in$ *First*(*args*). We now observe that the *First* sets are now complete, since nothing new is added at the third round.

Note that we can also sensibly define the *First* set not just of a single symbol but of an arbitrary sentential form $t = x_1 x_2 \cdots x_n$ ($n \geq 0$): just replace A by t in the definition on page 2. Once we have all the sets *First*(A), for any t we may easily compute *First*(t) as follows:

1. Whenever we have $a \in$ *First*(x_i) where $1 \leq i \leq n$, $x_1, \ldots, x_{i-1} \in E$, and $a \neq \varepsilon$, put $a \in$ *First*(t).
2. If $x_1, \ldots, x_n \in E$, put $\varepsilon \in$ *First*(t).

We will need certain such sets *First*(t) in order to construct the parse table.

Computing *Follow* sets

Follow sets are also calculated progressively; the algorithm makes use of *First* sets.

1. Begin with *Follow*(*S*) $= \{\$\}$ for the start symbol *S*, and *Follow*(*A*) $= \emptyset$ for all other nonterminals.
2. For each rule that can be presented as $A \rightarrow tBx_1 \cdots x_n$ for $n \geq 1$, and each i such that $x_1, \ldots, x_{i-1} \in E$, add all of *First*($x_i$), except ε, to *Follow*(*B*).
3. For each rule $A \rightarrow tB$, or $A \rightarrow tBx_1 \cdots x_n$ with $x_1, \ldots, x_n \in E$, add all of *Follow*(*A*) to *Follow*(*B*).
4. Repeat step 3 until all *Follow* sets stabilize.

This is generally rather trickier to carry out than the calculation of *First* sets. Let us see how it works for our example. In step 1 we set *Follow*(*shell*) $= \{\$\}$ and *Follow*(A) $= \emptyset$ for $A =$ *args*, *opts*, *files*. Step 2 only yields anything in the case of the production *args* $\rightarrow$ *opts files*: here we must add all of *First*(*files*) except ε to *Follow*(*opts*), thus file $\in$ *Follow*(*opts*). Moving on to step 3, we discover on the first round that \$ $\in$ *Follow*(*args*), and on the second round that \$ $\in$ *Follow*(*opts*) and \$ $\in$ *Follow*(*files*).

To summarize, the contents of the sets *First*(A) and *Follow*(A) for the grammar G are as given by the following table:

G	*First*	*Follow*
shell	command	$
args	option file ε	$
opts	option ε	$ file
files	file ε	$

Building parse tables

Given the *First* and *Follow* sets for a grammar, constructing a parse table is reasonably straightforward. Initially, every entry is empty; then for every production $A \longrightarrow t$ in the grammar:

- for each terminal a in $First(t)$, put $A \longrightarrow t$ in the table at row A, column a;
- if $\varepsilon \in First(t)$, then for each $b \in Follow(A)$ enter $A \longrightarrow t$ in row A, column b.

Applying this procedure to our example, we obtain the following parse table:

G_3	command	option	file	$
shell	command *args*			
args		*opts files*	*opts files*	*opts files*
opts		option *opts*	ε	ε
files			file *files*	ε

For a given grammar, the fact that this method succeeds in constructing a parse table with at most one production in each cell provides the final confirmation that our grammar is indeed $LL(1)$ and hence suitable for predictive parsing. If the grammar suffers from ambiguities, common prefixes or left recursion, this will typically show up in the fact that some cell contains two or more productions, so during parsing we will not know which of the competing productions to apply. (There are even some grammars not suffering from any of these three defects for which such clashes arise, though such grammars are unlikely to arise in practice.) Of course, if clashes do arise in the parse table, then *where* they arise can give us a useful hint on how we might "debug" the grammar.

Automatic lexer and parser generators

When any programming activity becomes repetitive, programmers naturally try to get the machine to do it for them. So it is with writing lexers and parsers: the formal theories of state machines and grammars give us sufficient insight to

write programs that will write our programs for us. The first lexer- and parser-writing programs were written around 1970; the best known being `lex` and `yacc` ("Yet Another Compiler Compiler"). Both of these generate C code and are still in use today. They have several descendants and variants, so for example the Linux systems here provide `flex` and `bison`. Other programming languages have equivalent tools: for instance, for Java we have the tools *JFlex* and *Java CUP*. More detailed information of these tools may be found at (respectively)

`www.jflex.de` `www.cs.princeton.edu/~appel/modern/java/CUP`

With tools like these, building a language processing system generally comprises three steps.

- Prepare a description of the tokens of the language. Give this to `lex`, or similar, which will write some code for a lexer.
- Write a grammar for the language. Feed this to `yacc`, or similar, which will write some code for a parser.
- Write some code to operate on a document once it has been read in.

These three pieces of code are then tied together and compiled to give the final program. Usually the first two contain some large tables, with small general engines to drive the lexing and parsing. They are often not very human-readable, but that is not their aim.

Exercises

Revisit some of the examples of grammars from earlier lecture notes, and compute the First and Follow sets, and hence the parse table, using the above method. Check these against the parse table given in the notes.

It is also quite instructive to try this out on some of the examples of non-$LL(1)$ grammars (e.g. ambiguous grammars), so that you can see where the clashes arise when we try to build the parse table.

Books

The following books may provide a useful supplement to the lecture notes for the second half of this thread (context-free grammars and parsing). They may be readily identified by the animals depicted on the cover.

Dragon book Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*.

Tiger book Appel, *Modern Compiler Implementation in* { *C, Java, ML* } .

Turtle book Aho and Ullman, *Foundations of Computer Science*.

John Longley 2002, Ian Stark 2001