41080 Theory of Computer Science
Assignment 2
Group Programming Assignment

Tom Golding: 14316196
Samuel Franks-Pearson: 14272177
Parser: LL (1) Parser

Table of Contents:

Introduction

This report discloses how the created program utilises the material from the subject, specifically discussing how the constructed parser works, and what methods were used to create it. Topics such as how the team effectively collaborated with one another will be discussed. The elaboration on the context of a syntax analyser and its function in a compiler as well as a formal specification of a PDA and how we extracted the necessary information from the specification will be discussed. Followed by sections regarding what particular parser was used, the theoretical computation model for each component, and how efficient the implemented algorithm is. Concluding with additional techniques, challenges, and acknowledgments.

Group Member Contributions

The group members of this group were Tom Golding (14316196) and Samuel Franks-Pearson (14272177). This report will generally refer to them as “the group” or “group members”.

The overall distribution of work was equal between both group members, with Tom focusing more on the coding side, (as this was more his specialty, and he had coded the first assignment in Java) and Samuel focused more on the theoretical side (as this was more his specialty, and had coded in Python for the first assignment).

Both group members also contributed equally to the writing of this report and agreed that no particular bias was shown to a specific individual.

Collaboration Methods

In order to collaborate effectively on the project, the group used a combination of GitHub and the Workspace tool on the EdStem website. EdStem was primarily used for coding at the same time when meeting in person or in a call, to work out the general concepts of the project. GitHub was used to keep track of individual progress of the project, with the main file being updated whenever newer and more successful code was implemented.

A shared Excel spreadsheet was used to make the first and follow sets, as well as the parse table, and Google docs was used to write out the report. The group primarily used Discord to communicate. The Github repository can be found here: https://github.com/ComanderPotato/Assignment-2-ToCs .

Group members also met a few times in person on the UTS campus to collaborate on the project.

Formal Definition of a Syntax Analyser

Carrying over from the previous assignment, we delve into the second phase of a compiler, known as syntax analysis or parsing. We’ve seen the capability of lexical analysis and its ability to identify inputted tokens and convert them into lexemes, it is a crucial part of a compiler however due to its limitations of regular expressions, a lexical analyser doesn’t have the ability to check syntax. This is where a syntax analyser displays its importance within the compiler, as

it takes the token stream from the lexical analyser and analyses the source code against the production rules of the grammar to detect errors. During the runtime of the syntax analyser it does two primary tasks, error detection as stated above, and generating a parse tree which is a graphical depiction of the derivation as the output of the phase. Furthermore, this phase relies heavily on a Context Free Grammar (CFG) and a PDA. The CFG must be left factored and have left-recursion removed, which is crucial for the PDA to recognize what rules to apply for the given stack top symbol and next read token.

Formal Specification of Pushdown Automata

A PDA consists of seven components, commonly known as a seven-tuple. These components include:

1) The set of states Q
2) The input alphabet Σ (sigma)
3) The stack alphabet Γ (gamma)
4) The transition function δ : $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$
5) The start state $q_0 \in Q$
6) The set of accept states $F \subseteq Q$
7) The initial stack top symbol $\bot \in \Gamma$

Which can be formally identified as PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \bot)$, given an string $w = w_1, w_2, \ldots, w_n \in \Sigma^n$, M accepts w if there exists a sequence of states$(r_0, r_1,...,r_n)$, $r_i \in Q$, and a sequence of strings $(S_0, S_1,..., S_n)$, $S_i \in \Gamma^*$, such that:

1) $r_0 = q_0$
2) $(r_i, w_{i+1}, a) = (r_{i+q}, b)$ and $S_i = at$, $S_{i+1} = bt$ which means to read a letter and a stack symbol → change its state and write a stack symbol
3) $r_n \in F$, which means to end with an accept state

So M rejects w if it doest not accept w

Extracting Information for the PDA

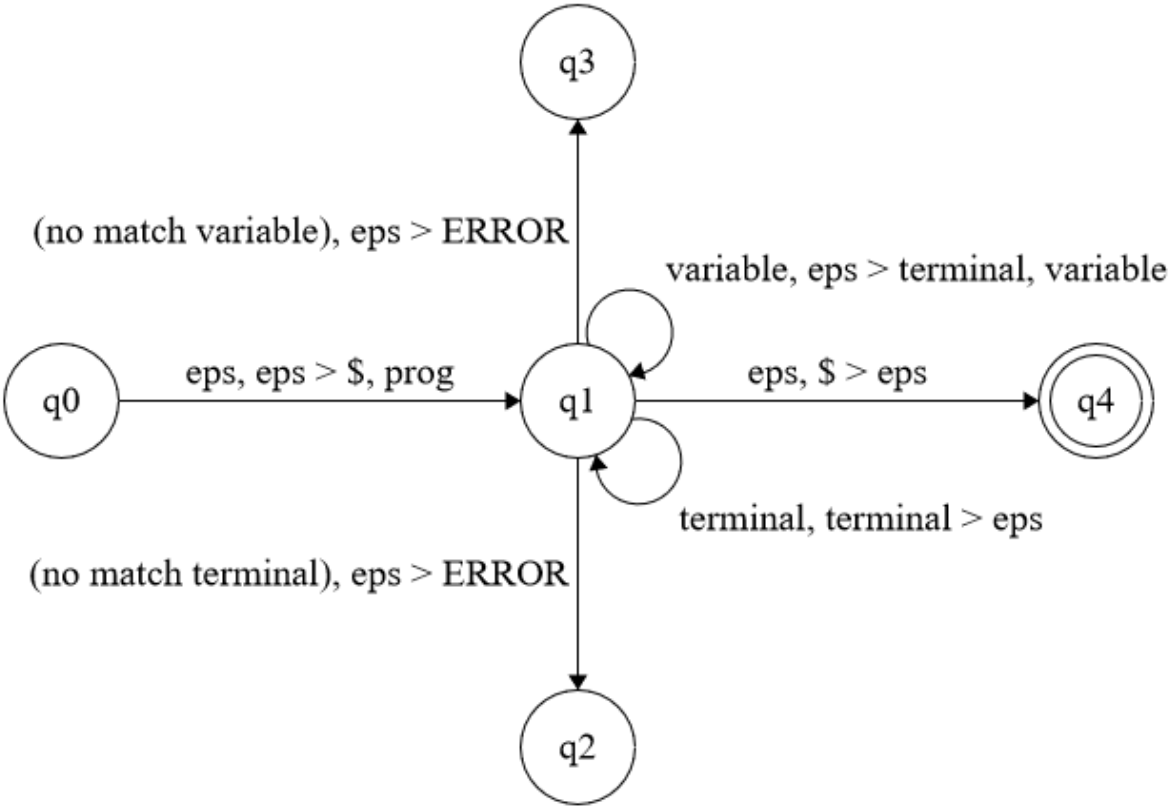
To extract the information needed for the PDA, the group collectively read through the specification provided. We started with identifying terminals and variables that would constitute our alphabet of Σ and Γ. Our input symbols were very straightforward, as they consisted almost entirely of reserved java keywords, apart from words that were user-generated such as ID, CHARLIT, and STRINGLIT. Our stack alphabet was also very straightforward as the variables were provided to us via the SimpleJava Grammar page on Ed, our stack alphabet is also inclusive of Σ. The transition functions were also provided via the SimpleJava Grammar page with already being left factored and had left-recursion removed. With these components established we were able to work on the initial stack symbol and states of our PDA. A PDA uses $ as the end-of-input stack symbol and accepts the input if the stack only contains that symbol. However, we decided to have our initial stack symbol be prog, and only ever accept the input if on completion of parsing the tokens the stack is empty. Pushing this symbol to the stack can be seen on line 249. Within our finite controller, there are two conditional checks to determine

whether a Syntax Exception will be thrown. The first one can be seen on line 266 which will throw an exception if there is no rule associated with the current stack top variable and token being read, this is our state q_1. Our second one can be seen on line 277 which will throw an exception if the current terminal on the top of the stack doesn't match with the token in the input stream, this is our state q_2 Our last condition can be found outside the finite controller, which can be seen on line 280 this is a conditional check that will fail if the input is empty on runtime, and if the stack isn't empty on completion, this is state q_0, which in turn is also our start state. Our final fail state is the finite controller as a whole, as, the token stream is never accepted during the loop and only on completion, this will be our q_3. For our accept state, there is only one, which on completion of the parser, the stack size is empty, this is state q_4.

Formal specification of our PDA

Given the information gathered from the above section, determining the components for our PDA has been made simple. The LL(1) parser used in our code can be represented as the finite automata as follows, for PDA M = (Q, Σ, Γ, , q_0, F, ⊥), where:

1) Q = { q_0, q_1, q_2, q_3, q_4 }
2) Σ ={ PLUS, MINUS, TIMES, DIVIDE, MOD, ASSIGN, EQUAL, NEQUAL, LT, LE, GT, GE, LPAREN, RPAREN, LBRACE, RBRACE, AND, OR,SEMICOLON, PUBLIC, CLASS, STATIC, VOID, MAIN, STRINGARR, ARGS, TYPE, PRINT, WHILE, FOR, IF, ELSE, DQUOTE,SQUOTE, ID, NUM, CHARLIT, TRUE, FALSE, STRINGLIT }
3) Γ = { prog, los, stat, whilestat, forstat, forstart, forarith, ifstat, elseifstat, elseorelseif, possif, assign, decl, possassign, print, type, expr, boolexpr, boolop, booleq, boollog, relexpr, relexprprime, relop, arithexpr, arithexprprime, term, termprime, factor, printexpr, charexpr, epsilon, terminal, and the input alphabet Σ }
4) = see grammar and parse table in appendix
5) q_0 = { q_0 }
6) F = { q_4 }
7) ⊥ = { prog }



This is by no means an accurate representation of the LL(1) parser system, but instead acts as a model for how the parser generally operates, as a PDA. With an empty stack, the end stack symbol is pushed in, as well as the prog variable. Then for every variable read, terminals or variables are pushed in, based on the rules. For every terminal read, it gets matched (not shown) and then removed from the stack. If no match is found for a variable or token, it throws an error, and the PDA ends. If everything ends up being empty, then the PDA pops the end-of-input symbol, and the input is accepted.

What Type of Parser was Used, and the Reasoning Behind this Decision

This assignment used an LL(1) parser, which is a type of Top-down parser. A general LL parser is referred to as an LL(k) parser, where k represents the number of tokens k that the parser looks ahead by. The primary reason for choosing a LL(1) parser was that group members felt that there was no need to read more than 1 token ahead. As the ruleset that was supplied had rules where the head of the rule only had one condition, the parser only needs to read one token to determine its next step.

Given the intuitive nature of creating a tree (that is, starting from the root, creating nodes off that root that act as “branches”, which then have their own child nodes), it made sense to use a type of top-down parser, as these start from the highest level (i.e. the root), and work down the tree (through the nodes). The grammar was written as a single parent terminal that was linked to more terminals and/or variables, which also suited a top-down parsing method. A type of

bottom-up parser could have been used, but it would have posed more of a challenge to understand how the rules would work using this method.

Another type of top-down parser is a recursive descent parser, however, the group chose not to use this as they had a much better understanding of LL(1) parsers, due to them being the primary focus of this course.

How the Parser Components were Constructed

When creating the Parser, the first step was to analyse the grammar, and work out what the first and follow sets were (see appendix). Once this was completed, a parse table was generated, which was used to formulate the rules that were later used in the code.

Next, the group looked through the supplied skeleton code, as well as the supplied information on how to return a parse tree. Several different methods were attempted, and initially the code used a loop that traversed over the input tokens, and read from hard-coded rules. After some experimentation with map tools and the supplied Pair class, a hash map was used to contain the rules, storing a list of symbols as the value, with a (Symbol, Token) Pair acting as a key. However, this original iteration of the hash map only had the first 2 rules implemented.

After meeting up a few times to discuss the logic and process the code should follow, it was worked out that while there would be a large hash map to start, the majority of the processing could be done within a small loop.

The first 5 rules were put into the hash map, using the first and follow sets as a guide to what tokens would be recognised. Then the small loop was worked on, and once it was able to pass the early test cases with just the first 5 rules, the group worked on implementing the rest of the rules into the loop. After this was done, 28/30 test cases were able to be passed. The two test cases that were still failing, for the most part, returned a correct parse tree but had a clump of epsilon tokens, that weren't meant to be in a particular spot. After working through these small bugs, the group was able to pass all test cases, in approximately 283 lines

The Theoretical Computation Model each Component of the Code uses

In the construction of the LL(1) Parser model, 3 main data structures were used to construct it. The first structure is the hash map.

```
/*rules with a Key/Value pair that consists of a pair for the
stack symbol and read token input and value that stores the list of symbols
for the associated rule  */
Map<Pair<Symbol, Token.TokenType>, List<Symbol>> rules = new HashMap<>();

List<Symbol> rule1 = new LinkedList<>(Arrays.asList(/*List of symbols for the rules */));
rules.put(new Pair<>(/*Stack top symbol*/,  /*Read token*/), rule2p1);
```

In the above code, the Map called "rules" is created using a key composed of a Symbol and Token.TokenType Pair, with the corresponding value being a List of Symbols (where a symbol is a tree node or a token). Then the rule is created. The first rule is called rule 1, and is a list of

symbols. The Arrays.asList method allowed us to create and fill the list in one line. The list is filled in reverse, so for rule 1, RBRACE is first in the list, and PUBLIC is last. The reasoning for this will be expanded upon in the stack data structure section below. Then rule 1 is put into the rules map using the .put method, which pairs the key, with rule 1 as the value.

The same process is repeated for the remaining rules. For the sake of simplicity, rules that had different possible transitions were referred to as “point rules”. That is, the first option of rule 2 would be rule 2 point 1 (written as rule2p1), and the second option would be rule2p2 etc. Ultimately, there are 31 rules, and roughly 62 separate transitions across the rules.

The next data structure used was Stack. There were several methods available that would do the same job, such as the recommended ArrayDeque structure, but the group went with Stack as it closely follows a proper PDA.

```
Stack<Pair<Symbol, TreeNode>> stack = new Stack<>();
Stack<TreeNode> parentStack = new Stack<>();
```

The code used two stacks, one which was the main stack, just called stack, had all the necessary grammar pushed to it. The parentStack was used by the code to pair the symbol and current parent as they were pushed to the stack following a correct rule match.

```
if(rules.containsKey(new Pair<>(stack.peek().fst(), token.getType()))) {
    rules.get(new Pair<>(stack.pop().fst(), token.getType())).forEach(child -> {
        stack.push(new Pair<>(child, parentStack.peek()));
    });
} else {
    throw new SyntaxException(tokens.toString());
```

This segment of code finite controller was primarily responsible for pushing rules to the stack. It would read what was at the top of the stack, as well as what token the loop was up to, and would attempt to match these values to ones found in the map. If it could find the relevant key, it would push each symbol in the list to the top of the stack. Due to the rules being put in backwards, the last symbol would be pushed in first, and the first symbol would be pushed in last, which leads to the desired result. If the map didn’t have a key that matched the search, it would throw a syntax error.

The third structure was the parse tree. This was mostly based on the skeleton code provided, so most of the work involving it was about making the tree and calling the provided methods. First, the parse tree was constructed, and a tree node prog was initialized using the TreeNode class.

```
ParseTree tree = new ParseTree();

TreeNode prog = new TreeNode(TreeNode.Label.prog, null);
```

```
else if(stack.peek().fst() != TreeNode.Label.epsilon){
    TreeNode variable = new TreeNode(TreeNode.Label.valueOf(stack.peek().fst().toString()), stack.peek().snd());
    stack.peek().snd().addChild(variable);
    parentStack.push(variable);
}
```

Once the code reached the while loop, assuming certain conditions were met, the tree would be added to the parse tree using code similar to what is shown above. This would be done by

creating a new tree node called variable, which was made of stack.peek().fst(), which takes the “first” part of the pair at the top of the stack, which becomes the tree node’s identifier (name), and then stack.peek().snd(), which would get the parent, so the code knew what this variable would be a child of. The tree then has the variable child added to it, and the variable becomes the new parent. This particular scenario handles situations where the current item at the top of the stack was a non-epsilon variable, but there was also code to handle adding epsilon terminals and terminals to the tree.

How the Code Implements the Algorithm Efficiently

The need to implement an algorithm that could efficiently incorporate the rules provided and construct the desired parse tree on any given input was imperative. We closely followed the rules known as BigO notation to do so. This form of mathematical notation is used to bind the growth of run time and describes the limiting behaviour of a function as the argument tends towards a specific value. The BigO of an algorithm is portrayed as having two key factors, time and space complexity. The time complexity of an algorithm is solely based on how long that algorithm takes to run in order to reach completion, whilst the space complexity of an algorithm is how much space (memory) an algorithm needs to run. For our algorithm's time complexity, our entire finite controller is within a 30-line while loop, which is linearly dependent on the input size, giving it a BigO(n) time complexity. Within this loop, there is a conditional loop that searches the hashmap when necessary which has a time complexity of BigO(1) so the worst-case time complexity of our algorithm is BigO(n).

In order to implement an efficient space complexity for our algorithm, we used various data structures such as a hashmap and stack. Our data structures and rules have all been initialized before the loop giving us a space complexity of BigO(1), within the loop we have two instances where we are using the data structures to supply space for our variables and terminals. Within the isVariable condition, we push our current variable to the parentStack, and when a given rule is identified, we push all of the symbols associated with the rule to the stack. These methods can be seen as both linear BigO(n), giving our space complexity a worst-case of BigO(n).

Based on this information, our algorithm performs efficiently given the time and memory management a PDA is expected to need. With both the time and space complexity giving a BigO(n) the only improvement that could be made was implementing an algorithm that has a logarithmic time of BigO(logn) which could potentially be done with an LL(k) parser.

Additional Techniques Used

The code used a variety of coding techniques in order to pass all the test cases. While most of these were already implemented in the skeleton code, there were a few additional coding techniques that were used to create a successful project. A hash map data structure was used to help store the grammar in an efficient and relatively concise manner. The group used the stack data structure to keep track of all the tokens and tree nodes that were being processed. This particular data structure was represented similarly to a list, but pushed and popped from

the “end” of the stack, so to counteract this, our rules were put in reverse order. The code also used a parent stack, referred to as a parentStack in the code, to keep track of which tree node was a parent, and more importantly, which tree node was the parent that was currently having children added to it. The Pair class was also not required to complete the task, however, the group chose to use it as it allowed them to better create the key for the key-value pair in the hash map. The .fst and .snd methods provided in the Pair class were also used to help construct the parse tree, and access the individual elements in the pair.

Improvements, Limitations, Constraints & Challenges

In terms of improvements, there are potentially some that could be made. The initialisation of the hash map could be put into a separate class, which then gets called, which would reduce the number of lines in the main class, as well as increase the readability of the code. It is also possible that there is a more efficient way of writing the while loop, although, at 32 lines, it would be hard to make it any more condensed without losing the required functionality.

The code is extremely efficient as possible in terms of processing, as it has a BigO(n) in both space and time complexity, which is only slower than BigO(logn) and BigO(1). However, given the parser type and additional tools used (LL(1) parser and a hash map), it would be extraordinarily challenging to make it any faster.

Generally, there weren’t many limitations for the project. The most notable one were the fact that, if a group chose to use the skeleton code (which our group did) then you were required to use the supplied naming conventions and methods, which took a bit to understand, as it was code from a third party that group members were seeing for the first time. Another limitation was that due to the way EdStem marks test cases, in certain scenarios, the group was unable to see if their code worked, as some test cases lead to an infinite loop, which meant EdStem wouldn’t mark any cases. This means certain points had to have an exit to an infinite loop hard coded before testing could occur.

The group was also constrained by their limited experience with data structures (specifically hash maps), which meant research had to be conducted on how to properly use these structures before they could be implemented in the project.

There were a few challenges posed by this project. Given the large amount of grammar, generating correct first and follow sets did pose a bit of a challenge, but after a few revisions, the group felt they had gotten them all correct. Starting out the project was also a little challenging, as although there was a lot of supplied information, it was a bit overwhelming, and it took some time to properly understand what the project required, as well as understand how certain classes worked, such as TreeNode and Pair. As mentioned above, there were also certain bugs in the code that made regular appearances, mainly causing the code to get stuck in an infinite loop. This would happen for the test case that required an error to be thrown, as the

code would try to look for something that wasn't actually there. While all bugs were ultimately fixed, they did pose some trouble to work out.

Acknowledgements

The website used to create the PDA seen in this report was Finite State Machine Designer, made by Evan Wallace in 2010. It can be found at https://madebyevan.com/fsm/ . It is free to use, and had no conditions attached to its usage.

The LL parser page provided by wiki https://en.wikipedia.org/wiki/LL_parser was additional help to properly understand the forms of a parser and provide extra information in regards to the characteristics of a LL(1) parser.

The Kopec Explains Software podcast #87
http://kopec.live/episode/5e1c5f99/compilers-and-interpreters gave some extra information in regards to compilers and interpreters.

The Neso Academy Youtube video on derivation trees
https://www.youtube.com/watch?v=u4-rpIlV9NI&ab_channel=NesoAcademy gave us some more in depth information regarding parse trees.

Further thanks to Youming Qiao, Luke Mathieson, and various tutors for in-person meet-ups and zoom calls.

Appendix

Note: This table uses all caps to represent certain terminals, such as ID and NUM. Int, boolean and char are also all shown separately, but in the code they are all registers as a TYPE token. The table also refers to the symbol “$” which is used to show an empty stack. However this is not used in the code, as the stack is checked using methods such as stack.empty().
The token “{“ and “}” is also referenced in the follow sets several times, and is also used to represent the beginning and end of the set, so only the last } denotes the end of the set, if there are two it means the right brace is also a member of the set. The same applies for left brace “{“

Variable First & Follow Sets

Variable	First	Follow
prog	{PUBLIC}	{$}
los	{while, for, if, ID, int, boolean, char, System.out.println, ;, eps}	{$, }}
stat	{while, for, if, ID, int, boolean, char, System.out.println, ;}	{$, }, while, for, if, ID, int, boolean, char, System.out.println, ;}
while	{while}	{$, }, while, for, if, ID, int, boolean, char, System.out.println, ;}
for	{for}	{$, }, while, for, if, ID, int, boolean, char, System.out.println, ;}
for start	{int, boolean, char, ID, eps}	{$,;}
for arith	{(, ID, NUM, eps}	{$, ;}
if	{if}	{$, }, while, for, if, ID, int, boolean, char, System.out.println, ;}
else if	{else, eps}	{$, }, while, for, if, ID, int, boolean, char, System.out.println, ;}
else?if	{else}	{$, {}
poss if	{if, eps}	{$, {}
assign	{ID}	{$, ;}
decl	{int, boolean, char}	{$, ;}
poss assign	{=, eps}	{$, ;}
print	{System.out.println}	{$, ;}
type	{int, boolean, char}	{$, ID}
expr	{(, ID, NUM, ‘, TRUE, FALSE}	{$, ;}
char expr	{‘}	{$, ;}
bool expr	{==, !=, &&, ||, eps}	{$, ;,)}
bool op	{==, !=, &&, ||}	{(, ID, NUM, ‘, TRUE, FALSE}
bool eq	{==, !=}	{(, ID, NUM, ‘, TRUE, FALSE}
bool log	{&&, ||}	{(, ID, NUM, ‘, TRUE, FALSE}

rel expr	{(, ID,NUM, ‘, TRUE, FALSE}	{$,), ;, ==, !=, &&, ||}
rel expr’	{<, <=, >, >=, eps}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=}
rel op	{<, <=, >, >=}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=}
arith expr	{(, ID, NUM}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=}
arith expr’	{+, -, eps}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=, +, -}
term	{(, ID, NUM}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=, +, -}
term’	{*, /, %, eps}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=, +, -, *, /, %}
factor	{(, ID, NUM}	{$,), ;, ==, !=, &&, ||, <, <=, >, >=, +, -, *, /, %}
print expr	{(, ID, NUM, “, TRUE, FALSE}	{$,)}

Terminal First Sets

+	+
-	-
*	*
/	/
%	%
=	=
=='	=='
!=	!=
<	<
>	>
<=	<=
>=	>=
(	(
)	)
{	{
}	}

&&	&&
||	||
;	;
public	public
class	class
static	static
void	void
main	main
String[]	String[]
args	args
int	int
boolean	boolean
char	char
System.out.println	System.out.println
while	while
for	for
if	if
else	else
"	"
(')	(')
ID	ID
TRUE	TRUE
FALSE	FALSE
CHARLIT	CHARLIT
NUM	NUM
STRINGLIT	STRINGLIT

Parse Table

Note: Boxes with no colour are the first sets, boxes in green are the follow sets, boxes in blue are found in both the first and follow sets. To save space in the table, System.out.print is written as PRINT.

	+	-	*	/	%	=	==	!=	<	>	<=	>=	(	)	{	}	&&	||	;
prog																			
los																2			2
stat																3			3
while																			
for																			
for start																			6
for arith													7						7
if																			
else if																9			9
else?if																			
poss if															11				
assign																			
decl																			
poss assign						14													14
print																			
type																			
expr													17						
char expr																			
bool expr							19	19									19	19	19
bool op							20	20									20	20	
bool eq							21	21											
bool log																	22	22	
rel expr													23						
rel expr'							24	24	24	24	24	24		24			24	24	24
rel op									25	25	25	25							
arith expr													26						
arith expr'	27	27					27	27	27	27	27	27		27			27	27	27
term													28						
term'			29	29	29		29	29	29	29	29	29		29			29	29	29
factor													30						
print expr													31						

	public	class	static	void	main	String[]	args	int	boolean	char	PRINT	while	for	if	else	"	(')	ID	TRUE	FALSE	CHARLIT	NUM	STRINGLIT	$
prog	1																							
los								2	2	2	2	2	2	2				2						2
stat								3	3	3	3	3	3	3				3						
while												4												
for													5											
for start								6	6	6								6						6
for arith																		7				7		7
if														8										
else if								9	9	9	9	9	9	9	9									9
else?if															10									
poss if														11										11
assign																		12						
decl								13	13	13														
poss assign																								14
print											15													
type								16	16	16														
expr																	17	17	17	17		17		
char expr																	18							
bool expr																								19
bool op																								
bool eq																								
bool log																								
rel expr																	23	23	23	23		23		
rel expr'																								24
rel op																								
arith expr																		26				26		
arith expr'																								27
term																		28				28		
term'																								29
factor																		30						
print expr																	31	31	31	31		31		