Compiler Project Part I

Jisen Ren, Zhiyi Cheng, Haoran Wang, Xiao Han

May 17, 2020

1 Input Parsing

To read the input from the JSON files, we resort to the third-party library `jsoncpp` provided by the teaching assistant. Located in the sub-directory `3rdparty`, this library could be included by adding the header file `json/json.h`. Using classes `Json::Value` and `Json::Reader`, the input file is parsed into key-value pairs, while each value could be converted into `string` form.

The subsequent step would be the lexical analysis for `kernel`. Through the process, we exploit the class `std::stringstream` and keep using its member function `peek` until the end of string. As the lexical rules are rather simple, we do not actually use automata. Instead, the several functions `read_*()` play the role. After calling this routine, the array `tokens` is established, each element of which is a `Token`, with members `type` and `name`.

It should be noticed that to handle multiple test cases in a single program, the `stringstream` object should be reinitialized after dealing with each case before starting the next. Otherwise, the results would be incorrect.

2 Syntax Analysis

For the sake of convenience, we perform some slight modifications to the grammar definition. Notice that in the original definition, P could simply be interpreted as a sequence of S strings delimited by semicolons, while a single S is divided into left-hand and right-hand expressions by an equal sign which appears nowhere else. To avoid cumbersome syntax entries while preserving the priority, we classify the binary arithmetic operators into two categories: the low-priority ones (which include $+$ and $-$) and high-priority ones (which include $*$, $/$, $//$ and $\%$)s. An extra non-terminal E is created for the sub-expressions connected by high-priority operators only. For each test case, a constant number could have only one type and should be treated as terminal. Therefore, we obey the following derivation rules, where each non-terminal is abbreviated into a single letter, $+$ denotes all low-priority binary operators, and $\times$ denotes all high-priority binary operators:

$$
\begin{aligned}
S'_R &\to R \\
S'_L &\to T \\
R &\to R + R \mid E \\
E &\to E \times E \mid (R) \mid T \mid S \mid \text{const} \\
T &\to \text{id}\langle C \rangle [A] \\
S &\to \text{id}\langle C \rangle \\
C &\to C, \text{int} \mid \text{int} \\
A &\to A, I \mid I \\
I &\to \text{id} \mid I + I \mid I + \text{int} \mid I \times \text{int} \mid (I)
\end{aligned}
$$

To perform syntax analysis, we decided to follow the SLR scheme, although the grammar above is not exactly an SLR one. Here, we design the automaton and its transition table:

State	Derivation Rules	Transition Edges
0	$S'_R \to \cdot R$ $R \to \cdot R + R \mid \cdot E$ $E \to \cdot E \times E \mid \cdot(R) \mid \cdot T \mid \cdot S \mid \cdot\text{const}$ $T \to \cdot\text{id}\langle C\rangle[A]$ $S \to \cdot\text{id}\langle C\rangle$	$R \to 1; E \to 2$ $(\to 3; T \to 4; S \to 5$ $\text{const} \to 6; \text{id} \to 7$
1	$S'_R \to R\cdot$ $R \to R \cdot +R$	$+ \to 8$
2	$R \to E\cdot$ $E \to E \cdot \times E$	$\times \to 9$
3	$E \to (\cdot R)$ $R \to \cdot R + R \mid \cdot E$ $E \to \cdot E \times E \mid \cdot(R) \mid \cdot T \mid \cdot S \mid \cdot\text{const}$ $T \to \cdot\text{id}\langle C\rangle[A]$ $S \to \cdot\text{id}\langle C\rangle$	$R \to 10; E \to 2$ $(\to 3; T \to 4; S \to 5$ $\text{const} \to 6; \text{id} \to 7$
4	$E \to T\cdot$	
5	$E \to S\cdot$	
6	$E \to \text{const}\cdot$	
7	$T \to \text{id} \cdot \langle C\rangle[A]$ $S \to \text{id} \cdot \langle C\rangle$	$\langle \to 11$
8	$R \to R + \cdot R$ $R \to \cdot R + R \mid \cdot E$ $E \to \cdot E \times E \mid \cdot(R) \mid \cdot T \mid \cdot S \mid \cdot\text{const}$ $T \to \cdot\text{id}\langle C\rangle[A]$ $S \to \cdot\text{id}\langle C\rangle$	$R \to 12; E \to 2$ $(\to 3; T \to 4; S \to 5$ $\text{const} \to 6; \text{id} \to 7$
9	$E \to E \times \cdot E$ $E \to \cdot E \times E \mid \cdot(R) \mid \cdot T \mid \cdot S \mid \cdot\text{const}$ $T \to \cdot\text{id}\langle C\rangle[A]$ $S \to \cdot\text{id}\langle C\rangle$	$E \to 13; (\to 3$ $T \to 4; S \to 5$ $\text{const} \to 6; \text{id} \to 7$
10	$E \to (R\cdot)$ $R \to R \cdot +R$	$) \to 14$ $+ \to 8$
11	$T \to \text{id}\langle\cdot C\rangle[A]$ $S \to \text{id}\langle\cdot C\rangle$ $C \to \cdot C, \text{int} \mid \cdot\text{int}$	$C \to 15$ $\text{int} \to 16$
12*	$R \to R + R\cdot$ $R \to R \cdot +R$	$+ \to 8$
13*	$E \to E \times E\cdot$ $E \to E \cdot \times E$	$\times \to 9$
14	$E \to (R)\cdot$	
15	$T \to \text{id}\langle C\cdot\rangle[A]$ $S \to \text{id}\langle C\cdot\rangle$ $C \to C\cdot, \text{int}$	$\rangle \to 17$ $, \to 18$
16	$C \to \text{int}\cdot$	
17	$T \to \text{id}\langle C\rangle \cdot [A]$ $S \to \text{id}\langle C\rangle\cdot$	$[\to 19$
18	$C \to C, \cdot\text{int}$	$\text{int} \to 20$
19	$T \to \text{id}\langle C\rangle[\cdot A]$ $A \to \cdot A, I \mid \cdot I$ $I \to \cdot\text{id} \mid \cdot I + I \mid \cdot I + \text{int} \mid \cdot I \times \text{int} \mid \cdot(I)$	$A \to 21$ $I \to 22$ $\text{id} \to 23; (\to 24$
20	$C \to C, \text{int}\cdot$	

21	$T \to \text{id}\langle C\rangle[A\cdot]$ $A \to A\cdot, I$	$] \to 25$ $, \to 26$
22	$A \to I\cdot$ $I \to I\cdot +I \mid I\cdot +\text{int} \mid I\cdot \times\text{int}$	$+ \to 27; \times \to 28$
23	$I \to \text{id}\cdot$	
24	$I \to (\cdot I)$ $I \to \cdot\text{id} \mid \cdot I + I \mid \cdot I + \text{int} \mid \cdot I \times \text{int} \mid \cdot(I)$	$I \to 29$ $\text{id} \to 23; (\to 24$
25	$T \to \text{id}\langle C\rangle[A]\cdot$	
26	$A \to A, \cdot I$ $I \to \cdot\text{id} \mid \cdot I + I \mid \cdot I + \text{int} \mid \cdot I \times \text{int} \mid \cdot(I)$	$I \to 30$ $\text{id} \to 23; (\to 24$
27	$I \to I + \cdot I \mid I + \cdot\text{int}$ $I \to \cdot\text{id} \mid \cdot I + I \mid \cdot I + \text{int} \mid \cdot I \times \text{int} \mid \cdot(I)$	$I \to 32; \text{int} \to 31$ $\text{id} \to 23; (\to 24$
28	$I \to I \times \cdot I$ $I \to \cdot\text{id} \mid \cdot I + I \mid \cdot I + \text{int} \mid \cdot I \times \text{int} \mid \cdot(I)$	$I \to 33$ $\text{id} \to 23; (\to 24$
29	$I \to (I\cdot)$ $I \to I\cdot +I \mid I\cdot +\text{int} \mid I\cdot \times\text{int}$	$) \to 34$ $+ \to 27; \times \to 28$
30	$A \to A, I\cdot$ $I \to I\cdot +I \mid I\cdot +\text{int} \mid I\cdot \times\text{int}$	$+ \to 27; \times \to 28$
31	$I \to I + \text{int}\cdot$	
32*	$I \to I + I\cdot$ $I \to I\cdot +I \mid I\cdot +\text{int} \mid I\cdot \times\text{int}$	$+ \to 27; \times \to 28$
33*	$I \to I \times I\cdot$ $I \to I\cdot +I \mid I\cdot +\text{int} \mid I\cdot \times\text{int}$	$+ \to 27; \times \to 28$
34	$I \to (I)\cdot$	
35	$S'_L \to \cdot T$ $T \to \cdot\text{id}\langle C\rangle[A]$	$T \to 36$ $\text{id} \to 37$
36	$S'_L \to T\cdot$	
37	$T \to \text{id}\cdot\langle C\rangle[A]$	$\langle \to 38$
38	$T \to \text{id}\langle\cdot C\rangle[A]$ $C \to \cdot C, \text{int} \mid \cdot\text{int}$	$C \to 39$ $\text{int} \to 16$
39	$T \to \text{id}\langle C\cdot\rangle[A]$ $C \to C\cdot, \text{int}$	$\rangle \to 40$ $, \to 18$
40	$T \to \text{id}\langle C\rangle\cdot[A]$	$[\to 19$

The analysis of right-hand expressions start from 0 and accept at 1, while that of right-hand expressions start from 35 and accept at 36. The states marked with an asterisk (*) might trigger shift-reduce conflict when encountering binary operators; however, in such case we would always choose to reduce by the rule where the $\cdot$ mark is at the end. For R and E, this guarantees that binary operators with the same priority are always calculated from left to right; in terms of I, since the calculation of I does not actually matter (in subsequent steps we merely need to print I in the way it looks like), even the priority is ignored.

In our source code, the basis class `Node` and the derived classes `Terminal`, `nTerminal` are used to store the grammar tree, while the function `Analyze`, the global variable `current` are responsible for syntax analysis. After the routine, the vector `Kernel` stores the root of the left-hand and right-hand side of each expression.

3 Code Generation

In this part we traverse the syntax tree generated by previous steps to generate the target C++ source code.

Summation Convention The Einstein's summation convention appears in a few cases. For example, the first sentence of the fifth case actually does the operation $A \leftarrow A + \alpha \times (B \times C)$. The summation convention appears for each "Indivisible" sub-expression where an index variable appears on the right side but not on the left side.

This is exactly where the non-terminal E shows its advantage. We first represent the right-hand expression by a sequence of E's connected by low-priority operators; for each "indivisible" expression E, if it is subject to the summation convention, a temporary variable is created which equals the shape of the left-side tensor; and then loops over all index variables perform the summation.

Determining Loop Ranges The ranges of `for` loops are determined by the tensor shape directly, since each loop variable in each case appear "alone" (not together with operators) at least once. When different indicators of the same variable's range occur (for example when $A\langle 8\rangle[i]$ and $B\langle 10\rangle[i]$ appear in the same expression), we choose the minimum of all.

Even if the range of each loop variable is determined, the array boundaries should be considered. Hence we perform a range check using `if` expression for each index expression consisting of not only loop variables but also arithmetic operators.

Type checking and casting is unnecessary, since the only thing we need to do with the expression is to print what it originally looks like.

Classes and Functions For code generation, we defined classes `TSRef`, `Sent` and `Func`. The class `Func`, representing the whole function, obtains some meta-information of the function from the JSON value, and contains an array of sentences `vector<Sent>` initialized by `Kernel`. The class `Sent`, representing one of the sentences, contains an array of tensor or scalar references `vector<TSRef>` initialized by one element of `Kernel`.

After initialization, each `Sent` and `Func` class synthesizes its index information. A `Sent` constructs the sentence by calling `build_Sent()`, while a `Func` generates the function signature by calling `build_sig()` and combines the sentences into a function by calling `build_Sent()` of each sentence.

Several auxiliary functions `get*()` and `print*()` are the assistance for debugging. In the main function, the function body is constructed by `Func body(Kernel,obj);` the target code string is generated by the function `body.getAnswer()`, stored in its return value.